

序1

——请接过这枝花

美国 Goldman Sacks 公司
朱天华

朱天华博士，1987 年毕业于中国科技大学近代物理系，1992 年和 1994 年于美国哥伦比亚大学获建筑学硕士学位（M.A.）和物理学硕士学位（M.Ph），1997 年获理论天体物理学博士学位（Ph.D）；现任美国 Goldman Sacks 公司利率产品资深策略专家。

在人类所制造的工具中，有的可以将人送上月球，有的可以驾驭原子的火焰，但只有计算机软件，才能透彻地反照出人类的心智。

软件是人类心灵和智慧在虚拟空间中的投射。软件的性能就是人类能力的扩展，它的活动就是人类心智活动的反映。软件直接表达出设计者对目标的理解，对用户的期待，以及对自己的定位。

然而，人类的软件却是问题多多。面对不断变化的性能要求，软件系统往往过于僵硬，过于脆弱，不易复用，很难维护。一个设计师驾驭软件系统的能力，就像他统治一个虚拟王国，或者主宰一个虚拟世界的能力一样；而这些软件设计中的问题，其实就是人类自身心灵和智慧的不足在镜子中的倒影。它反映出的，不仅仅是技术的不足，还包括了科学、艺术、心理和哲学的不足。

在人类思想的百花园里，有一朵奇葩散发出神奇的幽香，这就是由建筑工程学家 Christopher Alexander 发展出的模式理论。它涵盖科学、心理、艺术和哲学，不仅适用于建筑工程学，而且适用于软件工程学以及任何其他的工程学。Alexander 试图揭示的，是一条人类头脑与宇宙相互作用的永恒之道，它可以弥补发源于西方的现代科技、艺术和哲学的不足。

说到这里，忘记告诉你，这一散发着奇妙香气的曼妙花朵，嫁接自遥远的中国古代思想的园圃。

阎宏博士，凭借深厚的自然科学和中国传统哲学功底，以及多年来对模式理论的研究，将通过这本书向读者讲解软件设计模式的理论，它的简要历史，以及它与中国道家文化的渊源。这是一本实用的教材和手册，同时又是一本散发着思想火花的作品。当你把它放到书架上的时候，它应当占据一个独特的位置。

请接过这枝花。

VISIT…

LANZAROTE Caliente.COM



序 2

北京科蓝软件系统有限公司
郑仁寰

郑仁寰博士于 1986 年从英国曼彻斯特大学获得计算机科学博士学位，现任北京科蓝软件系统有限公司技术总监。

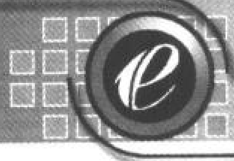
电子计算机科学是极为年轻的学科，几十年来，新理论、新技术层出不穷。计算处理能力的爆炸式增长使人类的“胃口”越来越大，所研制软件系统的复杂性等问题也与之俱来。20 世纪 60～70 年代，已有了“软件危机”的提法。针对这类问题，各种灵丹妙药应运而生。面向对象法（object-oriented approach）是近十余年的佼佼者。其理论经过 Smalltalk、C++、Java 等编程语言的实现以及现代用户界面的成功普及，可以说是通过了“实践的检验”。

设计模式来源于建筑学。他山之石，可以攻玉。这一理论，在软件行业的成功，恐怕是其发明人也始料不及的。设计模式把面向对象的理念发挥到了极至，对许多重复出现的问题，提出了既优雅又实际的解决方案。如今，作为高级软件设计师不了解设计模式，不能熟练地运用几个主要的模式，说其孤陋寡闻，恐怕也不过分。

Java 编程语言自 1995 年问世以来，其成功好像任何编程语言都无法媲美。生逢其时（互联网的兴起）固然是一方面的原因，而 Java 吸收总结了前人的经验教训，反映了最新技术（the state of the art），对其受到欢迎和采用，恐怕更为重要。例如，设计模式在 Java 的设计及功能中，有着举足轻重的作用。

十余年来，像西方一样，电脑新书的出版在国内市场也是爆炸式的，这些书出得仓促，寿命亦短（新版频繁，技术更新快）。令人遗憾的是，不少技术书或是语言枯燥，读起来令人味同嚼蜡；或是由于译者对所涉及的技术不甚通晓，加上校对错误，令人不知所云。近来，对中国软件业与印度相比不成气候，多有讨论，其对策往往涉及软件人才的培训、对英语的掌握、新书的翻译等问题。要想使中国软件业脱颖而出，没有一大批技术高超、爱业敬业、与时并进的软件人才，近乎缘木求鱼。夜郎自大、浅尝辄止，都是大忌。新技术的全面深入引进，至为关键。

阎宏博士有多年软件系统设计、培训的第一手经验，从事软件开发项目管理之余，将其对设计模式的深入理解、实践心得著述成书，以飨中文读者。本书资料翔实、取法新颖、技术前沿、举例生动，既可帮助有志于掌握设计模式的软件设计师做自学、参考书，又适用于高校及专业培训机构选为教材。



前言

设计模式和设计原则已经成为面向对象的编程（OOP），以及面向对象的设计（OOD）的最新进展。设计模式和设计原则可以帮助 Java 设计师针对日常系统设计工作所遇到的很多设计问题给出结构合理、易于复用、易于维护的示范答案。本书向国内的 Java 程序设计师介绍这一强大的工具。

作者简介



阎宏，1964 年出生于天津市。1987 年毕业于中国科技大学近代物理系，1990 年于中科院理论物理所获得硕士学位，1992 年获博士学位，翌年赴日本京都大学进行博士后研究工作。

作者曾于美国花旗银行（Citibank）、汤臣金融（Thomson Financial）、奥本海默基金（Oppenheimer）等处供职，进行了多年的软件开发、架构设计和技术管理工作。

欢迎读者与作者联系：JavaPatterns@yahoo.com，www.Web-EndsHere.com。

本书为何而写

促使作者写这本书的原因有很多，最主要的原因有以下几个。

第一，作者深深感到，虽然市场上有很多介绍面向对象技术的书籍，但是这些书籍中，有一些过于集中在理论层面，而有一些则过分地集中在具体层面上。

前者脱离软件工业的现状，对做具体工作的系统设计师和程序员的帮助十分有限。

后者由于拘泥于细节，很难帮助读者发展出全局的观念和视野。比如能够理解数据抽象化、继承、封装和多态性是非常重要的，但是仅仅知道这些并不能使设计师设计出具有更好的复用性和易于维护的系统。

通过前几年 UML 知识的普及，在读者群已经成熟的基础之上，如果能有一本贯穿理论和实践的教材性书籍，会对很多读者有益。

第二，作者深深感到，市场上没有一本以适合中国人的思维方式，针对中国读者群讲解软件设计与设计模式的书籍，这对大批的中国读者来说，十分不公平。

在西方的思想里面，分析一个结构直到把它分解为不可分割的单位，再加以数量化的研究和表述，是其典型的数理分析的套路。在西方科学文化中，不能够精确化和数量化的结构不受重视。

而在中国传统的文化里面，不能够精确化和数量化并不是研究的障碍。中国的阴阳思想可以灵活地描述任何的自然物、人造物和人类思想。

模式的理论有着深厚的中国道家思想渊源，设计模式实际上非常适合中国传统的思考方式。但是，很多中国的模式研究者们却似乎丢掉了自己的传统文化传承，像对待其他的舶来品一样，努力以西方的思维进行“重新”学习。

西方的作者往往由于自己对中国道家文化的理解水平所限，而在讲解模式理论的时候，有意无意地忽略它的道家背景。当这样的书籍充斥中国的模式图书市场的时候，中国的读者就会更加迷失在这一误区里面。

这使得作者感到一种沉重的责任感。

第三，市场上关于软件设计模式的书籍中确实有一部分是针对 Java 语言的，但是这些书籍在有些方面做得还不够。由于计算机图书的生命周期较短，就造成出版周期也很短，这使得一般的作者不愿意进行费时费力的调查研究工作。比如对于设计模式在 Java 语言 API 中的运用，以及设计模式是如何运用到真实的项目中去的，就较少有书籍做详细的介绍。

本书作者集中了很大的精力，对 Java 语言 API 中的设计模式做了详尽的综合、分析，并且从亲身参加的一些项目中抽取一些有代表性的问题，以专题讨论的方式介绍给读者。这可以使读者像亲临实地一样，体会设计模式是怎样应用到 Java 语言 API 中去的，以及是怎样在真实的软件项目中得到运用的。

本书为谁而写

Java 语言是现今最普及的纯粹 OOP 的编程语言，使用 Java 语言编程的程序师的平均素质也相对比较高。这些程序师往往不满足于只是实现程序功能的要求，他们常常想要在代码结构、编程风格，乃至解决问题的思考方式上不断进取和自我完善。设计原则与设计模式就是在大量的实践中总结和理论化之后优选的代码结构、编程风格及解决问题的思考方式。对模式的了解和掌握，是 Java 程序员提高自身素质的一个很好的方向。

这本书是 Java 程序师为 Java 程序员写的一本实用的设计原则和设计模式的自学教材和使用手册。

如果你是一位 Java 程序员，你以此为荣，并且希望能够了解 Java 语言 API 背后的设计思想和设计理念，以便更加了解这一语言，那么这本书就是为你而写的。

如果你是一位架构设计师，你以此为荣，并且希望能够进一步了解设计模式的起源，以及最知名的二十几个设计模式的用意、结构直到代码实现，以便能够改善自己的系统设计方案，那么这本书就是为你而写的。

如果你是一位设计模式的爱好者，你以此为荣，并且希望能够知道设计模式背后的设计原则和背后的哲学思想，那么这本书就是为你而写的。

本书的组织结构

本书的内容可以大体划分成两大部分：第一部分讲述了模式理论的起源、面向对象的设计原则；第二部分内容则是对具体的设计模式的讲解。

在第一部分内容中，本书首先对模式理论的起源做了简单但是比较全面的介绍。本书

特别对 Christopher Alexander 的模式理论与中国道家思想的联系做了哲学上的分析。然后，本书将几个最重要的面向对象的设计原则带给读者，这些原则都会在设计模式中具体体现出来。

在第二部分内容中，本书分别对最著名的几个创建模式、结构模式以及行为模式做了讲解。特别应当指出的是，本书的讲解有三大特点。

第一个特点，本书的讲解结合了具体的系统设计实例，比如单例模式和多例模式的讲解就结合了两个出现在真实的华尔街金融网站系统中的设计实例。读者可以将这些设计细节经过完善后使用到自己的系统设计中去。

第二个特点，本书的讲解结合了中国古代的神话传说、历史故事，内容贴切而生动活泼，可以给人以深刻的印象。

第三个特点，本书的讲解给出了大量的附图和可运行的代码，使读者可以一方面阅读文字，一方面欣赏插图，另一方面又可以将代码运行在 Java 环境中运行，并通过结果研究其结构。

怎样阅读本书

如果读者把本书当做一本手册，那么读者可以从书的任何一个章节开始阅读。本书的大多数章节都是与其他的章节相对独立、自成一体的，在少数依赖于其他章节的情况下，都会提醒读者首先阅读哪些章节。

如果将本书作为自学或教学的教材使用的话，则不一定要沿用模式的逻辑分类顺序。在下面就根据本书所包括的模式的难易、内在逻辑联系给出一个适合于自学成才和教学的顺序。

简单工厂模式（创建模式）：是最简单的模式，有助于引出工厂方法模式。

工厂方法模式（创建模式）：工厂方法模式在后面的许多模式都有应用，它在模式的讨论里屡见不鲜。了解工厂方法模式是了解所有模式的起点。

策略模式（行为模式）：策略模式在本书的讨论里频繁出现。早些理解策略模式有助于理解其他的模式。

装饰模式（结构模式）：装饰模式将是读者第一次接触到的模式的高雅之处。学会区别装饰模式和策略模式，也将是读者第一次学会区别看上去相似而实质不同的模式。

在学习了装饰模式之后，感兴趣的读者可以进一步阅读装饰模式在 Java I/O 中的应用，作为补充内容。

合成模式（结构模式）：合成模式在所有场合都有可能出现。合成模式经常与迭代子模式、责任链模式、解释器模式、访问者模式联合使用，因此，好好掌握合成模式有助于理解后面的有关模式。

迭代子模式（行为模式）：学习迭代子模式可以加深读者对合成模式的理解。本书还提供了关于 Java 语言对迭代子模式的支持的专题分析，供感兴趣的读者阅读。

模版方法模式（行为模式）：学习模版方法模式可以加深读者对策略模式以及简单工厂模式、工厂方法模式的理解。在迭代子模式里，模版方法模式经常被用到。

本书还提供了关于 Java Servlet 构造中的设计模式，主要是模版方法模式的专题研究。

抽象工厂模式（创建模式）：抽象工厂模式是创建模式里的第三个模式。读者回到这个模式，可以复习一下工厂方法模式。

在这一章后面的附录中，本书还给出了 Java 语言 API 中所使用的抽象工厂模式，供感兴趣的读者阅读。

建造模式（创建模式）：读者可以比较一下建造模式与抽象工厂模式的异同。

单例模式（创建模式）：单例模式经常用来实现抽象工厂模式。在单例模式的“相关模式”一节，对这一点有更详尽的描述。

本书为单例模式提供了一个专题研究章节，用来加深读者对这个重要模式在具体系统设计中的意义的了解。

多例模式（创建模式）：多例模式是对单例模式的推广。在学习了单例模式之后，学习多例模式，可以对隐藏在二者背后的设计思想有更深入的了解。

代理模式（结构模式）：读者在这里有机会学习代理模式怎样控制外界对一个对象的引用。代理模式也把读者引导到下一个模式——适配器模式。

本书利用了三章的篇幅，分别以代理模式的基本实现、智能引用代理和虚拟代理等具体种类详细地讲解了代理模式是怎样实现和应用到具体问题中去的。

适配器模式（结构模式）：读者可以学习和比较适配器模式与装饰模式、代理模式，以及后面的桥梁模式的异同。

在学习了适配器模式之后，感兴趣的读者可以进一步阅读适配器模式在 Java I/O 中的应用，作为补充内容。

桥梁模式（结构模式）：现在读者可以比较桥梁模式与适配器模式及代理模式的异同。

缺省适配模式（结构模式）：缺省适配模式是适配器模式的一种特例。在学习了普通的适配器模式之后，再学习缺省适配模式可以更好地体会两者的关系和区别。

调停者模式（行为模式）：学习调停者模式可以作为学习和理解观察者模式，以及 MVC（模型–视窗–控制，Model-View-Controller）设计的准备工作。

观察者模式（行为模式）：读者可以在学习中发现调停者模式是怎样在观察者模式里发挥作用，实现 MVC 这一经典设计的。

读者如果使用过 Java 语言的事件处理机制的话，实际上就已经使用过观察者模式，只是没有察觉而已。本书除了使用一章的篇幅讲解观察者模式本身之外，还使用了三章的篇幅讲解观察者模式在几个著名的 Java API 中的应用。

MVC 模式（架构模式）：MVC 模式包含合成模式、策略模式、观察者模式等，可以是这些设计模式的联合运用。

责任链模式（行为模式）：学习责任链模式，读者可以比较责任链模式和观察者模式、调停者模式是怎么处理调用的。

在这一章后面，本书还提供了一个专题章节，介绍使用责任链模式在 Java 语言里对“击鼓传花”游戏进行模拟的内容。

备忘录模式（行为模式）：读者学习备忘录模式，并讨论此模式是怎样实现 undo 和 redo 的，并比较下面要介绍的命令模式的处理方法。

命令模式（行为模式）：命令模式可以有很多种使用方法，其中之一与备忘录模式有关，因此，在备忘录模式后面介绍。

本书还对 Swing 库中命令的撤销做了介绍。

原始模型模式（创建模式）：这可能是创建模式中最复杂的一个。原始模型模式经常与命令模式一起使用，这就是为什么在命令模式后面介绍的原因。

作为补充阅读，本书还对 JavaBean 的串行化和并行化做了介绍。

状态模式（行为模式）：读者可以通过学习状态模式来理解一个对象改变状态的另一种方式。

本书利用状态模式对《聊斋志异》中的崂山道士的故事做了分析。

访问者模式（行为模式）：访问者模式经常与合成模式、迭代子模式一起联合使用。

本书在介绍访问者模式之前，对双重分派的概念做了介绍。

享元模式（结构模式）：这是所有常见的模式中最复杂的一个模式。这一模式的应用在下面的解释器模式里介绍。

解释器模式（行为模式）：这是一个复杂的模式。学习这一模式最好是在学习了享元模式以及访问者模式之后。

门面模式（结构模式）：这是最后学习的一个模式。在学习了解释器模式之后，门面模式就比较好学。因为讲解门面模式的环境常常与讲解解释器模式的环境是一样的。

以上仅供自学和教学者参考。

鸣谢

没有多位朋友的鼎力相助，本书不可能面世。首先，作者的同事 Bo Y. Yu 阅读了本书的大部分文稿，并提出了非常有用的意见。此外，Daniel Yu、David W. Liu、Wilson Liu、Richard Tang、Luo Ding、Tony Feng 也阅读了本书的部分文稿，并提出了宝贵意见，同时蒋涛先生、刘晓融先生也提出了很多好的意见。他们的意见都已经反映到本书的内容之中。

每一本书都是编辑与作者共同创造的成果。电子工业出版社的编辑对本书的面世做了大量细致入微的工作。

作者的爱妻冯君也曾数遍阅读本书，并在本书的写作期间给予了大量的支持。没有她自始至终的支持和帮助，这本书同样是不可能面世的。

在本书出版之际，本人对以上各位的支持致以最诚挚的感谢。

参考文献与进一步阅读

除特别注明的参考文献之外，本书的各个章节还使用了以下的参考文献。这些参考文献都是关于软件设计模式和其他模式理论的著作，可供读者做进一步的阅读。

[GOF95] E.Gamma，R. Helm，R. Johnson，and J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison–Wesley，1995

[GOF95Z] E.Gamma， R. Helm， R. Johnson，and J. Vlissides.《设计模式——可复用面向对象软件的基础》. 李英军，马晓星，蔡敏，刘建中译. 吕建审校. 机械工业出版社，2000

[VLISSIDES98] John Vlissides. Pattern Hatching. Addison–Wesley，1998

[GRAND98] Mark Grand. Patterns in Java. Vol. 1. John Wiley & Sons, Inc., 1998

[GRAND98-2] Mark Grand. Patterns in Java. Vol. 2. John Wiley & Sons, Inc., 1998

[GRAND98-3] Mark Grand. Patterns in Java. Vol. 3. John Wiley & Sons, Inc., 1998

[Cooper00] James W. Cooper. Java Design Patterns: A Tutorial. Addison-Wesley, 2000

[GOVONI99] Darren Govoni. Java Application Frameworks. Wiley Computer Publishing, 1999

[ALPERT98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. The Design Patterns Samlltalk Companion. Addison-Wesley, 1998

[STELTING02] Stephen Stelting, Olav Maassen. Applied Java Patterns. Prentice Hall, 2002

[J2EEPATT01] Deepak Alur, John Crupi and Dan Malks. Core J2EE Patterns. published by Prentice Hall, 2001

[METSKER02] Steven John Metsker. Design Patterns Java Workbook. published by Addison-Wesley, 2002

[ANTI99] William J. Brown, Hays W. McCormick III, Scott W. Thomas. AntiPatterns and Patterns in Software Configuration Management. Wiley, 1999

[SHALL02] Alan Shalloway and James R. Trott. Design Patterns Explained – A New Perspective on Object-Oriented Design. Addison – Wesley, 2002

[GOLD02] Brandon Goldfedder. The Joy of Patterns – Using Patterns for Enterprise Development. Addison – Wesley, 2002

[KIRK02] Kirk Knoernschild. Java Design – Objects, UML, and Process. Addison-Wesley, 2002

这些书籍都可以作为读者进一步阅读的方向。除了这些书籍之外，本书还向读者推荐以下的这些中文资料网站：

http://www-900.ibm.com/developerWorks/cn/index.shtml

http://www.yesky.com（程序方舟）

http://www.chinajavaworld.com

http://www.cn-java.com

http://www.csdn.net

http://www.smiling.com

http://www.WebEndsHere.com 本书的姊妹网站

目录

第一部分

第 1 章 模式的简史和形而上学 3
- 1.1 模式是什么 3
- 1.2 软件模式的简史 3
- 1.3 模式的起源 4
- 1.4 与道家思想的关系 5
- 1.5 软件的永恒之道 8
- 1.6 模式的要素 10
- 1.7 本书讲解模式的格式 11

第 2 章 统一建模语言 UML 简介 13
- 2.1 建造世界贸易中心 13
- 2.2 什么是 UML 14
- 2.3 UML 包括什么 15
- 2.4 类图 17
- 2.5 时序图 24
- 2.6 状态图 25
- 2.7 UML 及建模的工具 27

第二部分

第 3 章 软件的可维护性与可复用性 ... 33
- 3.1 软件系统的可维护性 33
- 3.2 系统的可复用性 37
- 3.3 老子论“不武” 40

第 4 章 “开－闭”原则（OCP） 41
- 4.1 什么是“开－闭”原则 41
- 4.2 怎样做到“开－闭”原则 41
- 4.3 与其他设计原则的关系 43
- 4.4 策略模式对“开－闭”原则的支持 45
- 4.5 在其他设计模式中的体现 46
- 4.6 一个重构做法的讨论 49

第 5 章 专题：Java 语言的接口 53
- 5.1 什么是接口 53
- 5.2 为什么使用接口 54
- 5.3 Java 接口常见的用法 56

第 6 章 专题：抽象类 61
- 6.1 什么是抽象类 61
- 6.2 抽象类的用途 61
- 6.3 基于抽象类的模式和原则 64
- 6.4 什么时候才应当使用继承复用 64

第 7 章 里氏代换原则（LSP） 69
- 7.1 美猴王的智慧 69
- 7.2 什么是里氏代换原则 70
- 7.3 里氏代换原则在设计模式中的体现 71
- 7.4 墨子论“取譬” 73
- 7.5 从代码重构的角度理解 74

第 8 章 依赖倒转原则（DIP） 85
- 8.1 为何而“倒转” 85
- 8.2 复用与可维护性的“倒转” 86
- 8.3 依赖倒转原则 86
- 8.4 怎样做到依赖倒转原则 88
- 8.5 Java 对抽象类型的支持 90
- 8.6 一个例子：账号、账号的种类和账号的状态 92
- 8.7 墨子论“取周” 95
- 8.8 依赖倒转原则的优缺点 95

第 9 章 接口隔离原则（ISP） 97
- 9.1 什么是接口隔离原则 97
- 9.2 一个角色隔离原则的例子 98
- 9.3 定制服务的例子 100

第 10 章 合成/聚合复用原则（CARP） 103
- 10.1 合成和聚合的区别 103
- 10.2 复用的基本种类 103
- 10.3 从代码重构的角度理解 105

第 11 章 迪米特法则（LoD） 109
- 11.1 迪米特法则的各种表述 109
- 11.2 狭义的迪米特法则 109

11.3 迪米特法则与设计模式......114
11.4 广义的迪米特法则......116
11.5 广义迪米特法则在类的设计上的体现......117
11.6 广义迪米特法则在代码层次上的实现......120
11.7 老子论“圣人之治”......120

第三部分

第 12 章 简单工厂（Simple Factory）模式......127
12.1 工厂模式的几种形态......127
12.2 简单工厂模式的引进......128
12.3 简单工厂模式的结构......134
12.4 简单工厂模式的实现......136
12.5 简单工厂模式与其他模式的关系......140
12.6 简单工厂模式的优点和缺点......142
12.7 简单工厂模式在 Java 中的应用......143
12.8 女娲抟土造人......146

第 13 章 工厂方法（Factory Method）模式......151
13.1 引言......151
13.2 工厂方法模式的结构......153
13.3 工厂方法模式在农场系统中的实现......157
13.4 关于工厂方法模式的实现......162
13.5 Java 语言中工厂方法模式的例子...165
13.6 工厂方法模式与其他模式的关系...168
13.7 女娲举绳造人......170
13.8 其他的例子......170

第 14 章 抽象工厂（Abstract Factory）模式......179
14.1 引言......179
14.2 问题......181
14.3 抽象工厂模式的结构......184
14.4 在什么情形下应当使用抽象工厂模式......189
14.5 抽象工厂模式在农场系统中的实现......191
14.6 抽象工厂模式的另一个例子......196
14.7 “开－闭”原则......197
14.8 相关的模式与模式的实现......198
14.9 女娲造万物的故事......200
14.10 附录：Java AWT 的 Peer 架构与抽象工厂模式......206

第 15 章 单例（Singleton）模式......209
15.1 引言......209
15.2 单例模式的结构......210
15.3 在什么情况下使用单例模式......215
15.4 单例类的状态......216
15.5 一个实用的例子：属性管理器......217
15.6 Java 语言中的单例模式......223
15.7 专题：不完全的单例类......226
15.8 相关模式......227
15.9 附录：双重检查成例的研究......230
15.10 给读者的一点建议......234

第 16 章 专题：单例模式与 MX 记录......235
16.1 问题与解决方案......235
16.2 目录服务与 MX 记录......237
16.3 JNDI 架构介绍......237
16.4 如何使用 JNDI 编程......238
16.5 系统设计......241
16.6 讨论......246

第 17 章 专题：多例（Multiton）模式与多语言支持......253
17.1 引言......253
17.2 多例模式......254
17.3 多语言项目的设计......258

第 18 章 专题：序列键生成器与单例及多例模式......267
18.1 问题......267
18.2 将单例模式应用到系统设计中......269
18.3 将多例模式应用到系统设计中......281
18.4 讨论......284

第 19 章 建造（Builder）模式......285
19.1 引言......285
19.2 建造模式的结构......286

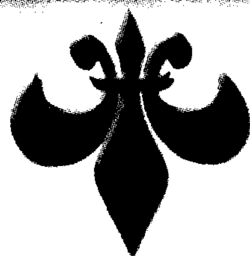
19.3 建造模式的活动序列......................290
19.4 建造模式的实现..............................291
19.5 众神造人的神话故事......................295
19.6 JavaMail 中的建造模式..................296
19.7 一个发送邮件的例子......................298
19.8 在什么情况下使用建造模式..........308
19.9 建造模式与其他模式的关系..........309
第 20 章 原始模型（Prototype）模式..317
20.1 引言..317
20.2 变量、对象以及对象的引用..........318
20.3 Java 对象的复制............................319
20.4 原始模型模式的结构......................323
20.5 模式的实现：深复制和浅复制......327
20.6 孙大圣的身外身法术......................329
20.7 在什么情况下使用原始模型模式..340
20.8 原始模型模式的优点和缺点..........341
20.9 原始模型模式与其他模式的关系...341
第 21 章 专题：JavaBean 的“冷藏”和“解冻”.............................345
21.1 什么是“冷藏”和“解冻”..........345
21.2 什么可以“冷藏”..........................346
21.3 Beans.instantiate()方法....................348
21.4 怎样在 JSP 中使用 Beans.Instantiate()方法....................353
21.5 与装饰模式的关系..........................354
第四部分
第 22 章 适配器（Adapter）模式........359
22.1 引言..359
22.2 类的适配器模式的结构..................360
22.3 类的适配器模式的效果..................361
22.4 对象的适配器模式的结构..............362
22.5 在什么情况下使用适配器模式......364
22.6 Iterator 与 Enumeration..................364
22.7 利用适配器模式指方为圆..............367
22.8 适配器模式在架构层次上的应用..370
22.9 关于模式实现的讨论......................373
22.10 适配器模式与相关的模式............373
第 23 章 缺省适配（Default Adapter）模式...379
23.1 鲁智深的故事..................................379
23.2 WindowAdapter 是缺省适配模式..381
23.3 模式的结构......................................384
23.4 在什么情况下使用本模式..............385
23.5 模式的实现......................................385
23.6 J2SE 中的缺省适配模式................386
23.7 一个例子..387
第 24 章 专题：XMLProperties 与适配器模式.............................391
24.1 引言..391
24.2 SAX2 浏览器..................................393
24.3 开始 XMLProperties 项目..............398
第 25 章 合成（Composite）模式........409
25.1 对象的树结构..................................409
25.2 介绍合成模式..................................411
25.3 安全式和透明式的合成模式..........412
25.4 安全式的合成模式的结构..............413
25.5 透明式的合成模式的结构..............416
25.6 合成模式的实现..............................419
25.7 道士的故事......................................420
25.8 一个绘图的例子..............................421
25.9 AWT 库中的例子............................428
25.10 合成模式与其他模式的关系........429
第 26 章 装饰（Decorator）模式........435
26.1 引言..435
26.2 装饰模式的结构..............................435
26.3 装饰模式应当在什么情况下使用..441
26.4 孙大圣的故事..................................441
26.5 使用装饰模式的优点和缺点..........442
26.6 模式实现的讨论..............................443
26.7 装饰模式与其他模式的关系..........446
26.8 实例：GrepReader..........................447
26.9 一个例子：发票系统......................451

26.10 附录：关于适配器模式与装饰模式的对话 464
第 27 章 专题：设计模式在 Java I/O 库中的应用 469
27.1 引言 469
27.2 Java I/O 库的设计原则 471
27.3 装饰模式的应用 472
27.4 半透明的装饰模式 480
27.5 适配器模式的应用 483
27.6 从 byte 流到 char 流的适配 489
第 28 章 代理（Proxy）模式 495
28.1 引言 495
28.2 代理的种类 496
28.3 代理模式的结构 498
28.4 代理模式的时序 501
28.5 Java 2.0 对代理模式的支持 502
28.6 高老庄悟空降八戒 506
28.7 代理模式的优点和缺点 508
28.8 代理模式的实现 509
28.9 代理模式与其他模式的关系 509
第 29 章 专题：智能引用代理 513
29.1 问题 513
29.2 系统设计 513
29.3 系统的时序 516
29.4 系统的源代码 516
第 30 章 专题：虚拟代理的例子 521
30.1 问题 521
30.2 系统的要求 521
30.3 系统的设计 522
30.4 系统的时序 522
30.5 系统的源代码 523
第 31 章 享元模式（Flyweight Pattern） 527
31.1 引言 527
31.2 单纯享元模式的结构 528
31.3 复合享元模式的结构 532
31.4 模式的实现 537
31.5 一个咖啡摊的例子 543
31.6 咖啡屋的例子 548
31.7 享元模式应当在什么情况下使用 553
31.8 从代码重构的角度看享元模式 553
31.9 享元模式的优点和缺点 555
31.10 相关的模式 555
第 32 章 门面（Facade）模式 561
32.1 引言 561
32.2 门面模式的结构 563
32.3 门面模式的实现 563
32.4 在什么情况下使用门面模式 564
32.5 迪米特法则（LoD） 564
32.6 一个例子 565
32.7 使用门面模式的设计 568
32.8 Session 门面模式 572
第 33 章 桥梁（Bridge）模式 575
33.1 引言 575
33.2 桥梁模式的结构 577
33.3 Java 语言中的 Peer 架构 579
33.4 驱动器和 JDBC 驱动器 583
33.5 从重构的角度考察 584
33.6 桥梁模式和其他模式的关系 590
33.7 八戒“转世投胎”的故事 592
33.8 关于桥梁模式的实现 594
33.9 在什么情况下应当使用桥梁模式 595
第五部分
第 34 章 不变（Immutable）模式 603
34.1 引言 603
34.2 不变模式的结构和实现 603
34.3 不变模式在 Java 语言中的应用 605
34.4 不变模式的优点和缺点 606
34.5 不变模式与享元模式的关系 606
34.6 一个例子：复数类 607
第 35 章 策略（Strategy）模式 621
35.1 引言 621
35.2 模式的实现 623
35.3 Java 语言内部的例子 624
35.4 排序策略系统 626

35.5 一个例子：图书折扣的计算........627
35.6 在什么情况下应当使用策略模式........632
35.7 策略模式的优点和缺点........632
35.8 策略模式与其他模式的关系........633
35.9 设计原则的讨论........635
35.10 诸葛亮的锦囊妙计........635
第 36 章 模版方法（Template Method）模式........641
36.1 引言........641
36.2 模版方法模式的结构........641
36.3 “好莱坞原则”........643
36.4 一个例子........644
36.5 继承作为复用的工具........647
36.6 Java 语言里面使用过的模版方法模式........648
36.7 模版方法模式中的方法........650
36.8 模版方法模式在代码重构中的应用........651
36.9 重构的原则........655
36.10 西天取经的八十一难........656
第 37 章 专题：Servlet 技术中的模式........659
37.1 Servlet 技术介绍........659
37.2 模版方法模式的使用........661
37.3 观察者模式的应用........665
第 38 章 观察者（Observer）模式........671
38.1 引言........671
38.2 观察者模式的结构........672
38.3 另一种实现方案........676
38.4 Java 语言提供的对观察者模式的支持........679
38.5 菩萨的守瓶龟........686
38.6 Java 中的 DEM 事件机制........687
38.7 观察者模式与其他模式的关系........688
第 39 章 专题：观察者模式与 AWT 中的事件处理........691
39.1 引言........691
39.2 DEM 的结构........691
第 40 章 专题：观察者模式与 SAX2 浏览器........705
40.1 引言........705
40.2 SAX2 是怎么工作的........706
40.3 ContentHandler 接口........706
40.4 怎样实现 ContentHandler 接口........708
40.5 怎样使用 ContentHandler........710
40.6 缺省适配模式与 DefaultHandler........711
40.7 简单工厂模式的应用........716
第 41 章 专题：观察者模式与 Swing 定时器........717
41.1 为什么需要定时器........717
41.2 Utility 定时器与 Swing 定时器的区别........717
41.3 使用 Swing 定时器的方法........719
41.4 观察者模式的应用........720
41.5 单例模式的应用........724
第 42 章 专题：MVC 模式与用户输入数据检查........729
42.1 什么是 MVC 模式........729
42.2 MVC 是架构模式........731
42.3 MVC 模式在 J2EE 技术中的应用........733
42.4 从代码重构的角度查看........735
42.5 用户输入检查与 MVC 模式........736
42.6 SQL 注射........738
第 43 章 迭代子（Iterator）模式........743
43.1 引言........743
43.2 迭代子模式的结构........745
43.3 迭代子模式的实现........755
43.4 迭代子模式的优点和缺点........757
43.5 一个例子........757
43.6 迭代子模式与其他模式的关系........763
第 44 章 专题：Java 对迭代子模式的支持........769
44.1 Java 聚集........769
44.2 Java 聚集中的迭代子........771
44.3 java.util.Iterator 接口........772
44.4 Java 迭代子接口 ListIterator........775

44.5 Enumeration 与 Iterator 780
44.6 其他相关模式....................782
第 45 章 责任链（Chain of Responsibility）模式....................785
45.1 从击鼓传花谈起....................785
45.2 责任链模式的结构....................785
45.3 纯的与不纯的责任链模式....................788
45.4 Java 1.0 版的 AWT 事件处理机制....................789
45.5 《红楼梦》中击鼓传花的故事....................790
45.6 Java 系统的解....................792
45.7 责任链模式的实现....................798
45.8 对象的树结构....................799
45.9 DHTML 中的事件处理....................800
45.10 责任链模式与其他模式的关系....................804
第 46 章 专题：定时器与击鼓传花....................809
46.1 线程....................809
46.2 定时器....................810
46.3 击鼓传花....................812
第 47 章 命令（Command）模式....................823
47.1 引言....................823
47.2 命令模式的结构....................824
47.3 玉帝传美猴王上天....................827
47.4 Java 语言内的例子：AWT 的事件处理....................828
47.5 一个例子：创世纪系统....................828
47.6 一个例子：AudioPlayer 系统....................833
47.7 增加宏命令功能....................839
47.8 模式的实现....................841
47.9 在什么情况下应当使用命令模式....................842
47.10 使用命令模式的优点和缺点....................843
47.11 命令模式与其他模式的关系....................843
第 48 章 专题：Swing 库中的命令撤销和恢复....................853
48.1 在视窗系统中使用命令模式....................853
48.2 Swing 的基本撤销功能....................853
48.3 一休论禅的文字框....................861
第 49 章 备忘录（Memento）模式....................869
49.1 备忘录模式的结构....................869
49.2 备忘录模式的白箱实现....................870
49.3 双重接口及其在 Java 语言中的实现....................875
49.4 备忘录模式的黑箱实现....................878
49.5 负责人角色的增强....................883
49.6 备忘录模式与多重检查点....................884
49.7 “自述历史”模式....................890
49.8 “假如”协议模式....................894
49.9 备忘录模式与其他模式的关系....................895
49.10 备忘录模式的应用....................896
49.11 使用备忘录模式的优点和缺点....................898
49.12 观世音甘泉活树的故事....................899
第 50 章 状态（State）模式....................907
50.1 引言....................907
50.2 状态模式的结构....................907
50.3 曾侯乙编钟的描述....................909
50.4 状态模式的效果....................910
50.5 在什么情况下使用状态模式....................911
50.6 关于模式实现的讨论....................911
50.7 一个状态模式的应用例子：TCP....................912
50.8 状态模式在绘图软件中的应用....................914
50.9 用户登录子系统....................915
50.10 状态模式与策略模式的区别....................921
第 51 章 专题：崂山道士与状态模式....................923
51.1 有状态的墙....................923
51.2 异常逻辑....................923
51.3 从代码重构的角度考察....................924
第 52 章 专题：单分派和多分派....................929
52.1 分派的概念....................929
52.2 双重分派....................933
52.3 未来的 Java 语言....................940
第 53 章 访问者（Visitor）模式....................945
53.1 引言....................945
53.2 访问者模式的结构....................947

53.3 系统的时序图......953
53.4 在什么情况下应当使用访问者模式......954
53.5 使用访问者模式的优点和缺点......955
53.6 访问者模式的实现......956
53.7 电脑专卖系统：问题与对象......960
53.8 电脑专卖系统的访问者模式设计......962
53.9 与访问者模式有关的模式......976
第 54 章 解释器（Interpreter）模式......979
54.1 引言......979
54.2 解释器模式的结构......980
54.3 一个示意性的实现......981
54.4 有关模式......991
第 55 章 调停者（Mediator）模式......993
55.1 引言......993
55.2 调停者模式的结构......996
55.3 调停者模式的实现......1001
55.4 迪米特法则（LoD）......1002
55.5 调停者模式的优点和缺点......1002
55.6 调停者模式的使用条件......1003
55.7 与调停者模式有关的模式......1004
附录 A 设计模式一览表......1007
A.1 创建模式......1007
A.2 结构模式......1009
A.3 行为模式......1011
附录 B UML 图标及其 Java 实现一览表......1015
附录 C 中英术语对照表......1017



第一部分　UML 和模式介绍

这一部分包括两个内容：模式理论背景的介绍和 UML 的简介。

“模式的简史和形而上学”一章简单地讨论了模式理论的背景、它与中国道家思想的渊源关系，以及模式理论的基础概念。

“统一建模语言 UML 简介”一章简单地介绍了 UML 的背景、基本概念，以及后面将要用到的几种 UML 图。

第 1 章　模式的简史和形而上学

一个围棋下得好的人知道，好的“形”对于围棋非常重要。形是棋子在棋盘上的几何形状的抽象化。形就是模式（Pattern），也是人脑把握和认识外界的关键。人脑处理模式的能力也非常高超，人可以在几百张面孔中一下子辨认出所熟悉的脸来，就是一个典型的例子。模式化的过程是把问题抽象化，在忽略掉不重要的细节后，发现问题的一般性本质，并找到普遍适用的解决方案的过程。

1.1　模式是什么

现代科学和工程学能够发展到今天，有赖于规则的制定，模式的研究也不例外。在讨论模式之前，必须对“模式”这个词加以界定，以规范后面的讨论和研究。

简而言之，人们在自己的环境中不断发现问题和寻找问题的解决方案的时候，发现有一些问题及其解决方案不断变换面孔重复出现，但在这些不同的面孔后面有着共同的本质，这些共同的本质就是模式。

那么，模式是不是在某种环境下对某个问题的答案呢？

这不完全对。模式所描述的问题及问题的答案都应当是具有代表性的问题和问题的答案。所谓具有代表性，就是说它以不同的形式重复出现，允许使用者举一反三，将它应用到不同的环境中去。为了与其他人交流，通常还要求给这个问题和问题的答案一个名字。

不知道读者会不会注意到，上面所使用的“代表性”、“举一反三”等词汇，是中国人特有的表达方式。这样的词汇翻译成任何一种西方语言，都需要浪费很多口舌加以解释。这是因为，模式其实非常适合中国人的思维方式，而不是西方人的思维方式。读者在阅读了本章之后，就会对这一点有更深的认识。

1.2　软件模式的简史

在面向对象的编程中使用模式化方法研究的开创性著作是文献[GOF95]（中译本[GOF95Z]）。这本书的四位作者通常被戏称为“四人帮”（Gang of Four，或 GoF）。这本书发表于 1995 年，而模式理论被引进到软件设计界的历史要稍早一些。

设计模式在软件设计行业中的起源可以追溯到 1987 年。那时，Ward Cunningham 和 Kent Beck 在一起用 Smalltalk 做设计用户界面的工作。他们决定使用 Alexander 的理论发展出一个有五个模式的语言来指导 Smalltalk 的新手，因此他们写成了一篇“Using Pattern Languages for Object-Oriented Programs（使用模式语言做面向对象的程序）”的论文（发表

于 OOPSLA'87 in Orlando)。

在那以后不久，Jim Coplien 开始搜集 C++语言的成例（idioms）。成例是模式的一种，这些 C++成例发表在 1991 年出版的《Advanced C++ Programming Styles and Idioms（高级 C++编程风格和成例）》一书中。

从 1990 年到 1992 年，“四人帮”的成员开始他们搜集模式的工作。关于模式的讨论和工作会议则一再举行。

在 1993 年 8 月，Kent Beck 和 Grady Booch 主持了在科罗拉多的山区度假村召开的第一次关于模式的会议。模式研究的主要人物都参加了这次会议，包括 Jim Coplien，Doug Lea，Desmond D'Souze，Norm Kerth，Wolfgang Pree 等。

在那以后不久，“四人帮”的《Design Patterns》一书就发表了。

此书发表之后，参加模式研究的人数呈爆炸性增长，被确定为模式结构的数目也呈爆炸性增长。编程模式语言大会（Pattern Languages of Programming，或者 PLoP）每年一次定期在美国举行，大会的论文也汇编成书，公开发表为[PLOP95]、[PLOP96]、[PLOP98]、[PLOP99]。

模式也不断地被应用到软件工程的各个方面。在诸如开发组织、软件处理、项目配置管理等方面，都可以看到模式的影子，但至今得到了最深研究的仍是设计模式和代码模式。

1.3 模式的起源

有趣的是，模式的研究并非起源于软件工程行业。

建筑的永恒之道

模式的研究起源于建筑工程设计大师 Christopher Alexander 的关于城市规划和建筑设计的著作。尽管他的著作是针对城市规划和建筑设计的，但是作者的观点实际上适用于所有的工程设计领域，包括软件开发设计领域。

Alexander 在他的著作中指出，使用现在的设计方法所设计出的建筑物，不能满足所有工程设计的基本目的：改善人类的条件。Alexander 想要发明的建筑结构，是能使人类在舒适和生活质量上受惠的建筑结构。他得出的结论是，设计师必须不断努力，以创造出更加适合所有的住户、用户和他们的社区的结构，从而满足他们的需要。

Alexander 的著作有：[ALEX64]、[ALEX75]、[ALEX77]、[ALEX79]，后三本都有中译本。如果读者对他的著作感兴趣的话，本书推荐读者首先阅读[ALEX79]或者中译本[ALEX79Z]，也就是《建筑的永恒之道》一书。

在这些著作里，Alexander 描述了一些他认为是永恒的、适合于任何工程学科的设计原则。这些原则是建立在下面的三个基本概念上的：质、门、道。

论无名的质（QWAN）

在 Alexander 的论述中，质（The Quality）或无名的质（The Quality Without a Name，



或简称为 QWAN）处在核心的地位上。

Alexander 认为，所有的生物、有用之物均包含有如下的“质”：自由性，整体性，完备性，舒适性，和谐性，可居住性，持久性，开放性，弹性，可变性，以及可塑性。QWAN 使人感到充满活力，给人以满足感，并最终改善人类的生活。

论门

“门”（The Gate）是通向“质”的管道。“门”是通过一个普遍的模式语言来体现的，这个模式语言使设计师能够创建多种形式的设计，以满足多方面的需求。“门”是普遍存在的，是这些模式之间的关系或是模式的“以太”，充满着一个特定的域。

论道

“道”（The Way）又称做“永恒的道”（The Timeless Way）。

利用“道”，从“门”演化到“质”的过程，就是把一些特定的模式按照一定的顺序应用到系统设计上的循序渐进的过程。Alexander 把从“门”到“质”的过程比喻为胚胎发育过程。胚胎发育过程的特殊之处是整体先于部分而存在，整体通过分裂来产生部分。通过追寻“道”，可以通过“门”到达“质”。Alexander 认为这个过程就是任何一种工程设计的发展过程。

1.4　与道家思想的关系

阅读过 Alexander 的《建筑的永恒之道》一书的人们无不为此书的哲理性所打动。人们将这本书比喻为一本使用建筑学的例子讲解哲学的书，或者一本以哲学的风格讲解建筑学的书。Alexander 本人则一再强调，模式理论是一个完整的、不可分割的整体，不能够将他的理论概括成为一套方法论，而不理会他的哲学观点。

现在，模式的理论回到了道家思想的故乡。遗憾的是，黑眼睛黑头发的道家传人中，有多少人还能够依稀辨认出模式理论的哲学观点的道家背景呢？

强烈的道家观点

Alexander 的哲学观点有很深的道家渊源。

University of Texas at San Antonio 数学系的 Nikos A. Salingaros 博士总结了 Alexander 的工作[SALIN]，他说：“Alexander 的建筑学著作发展出了一套关于自然和生命的哲学……Alexander 认为宇宙是一个和谐的整体，包含了感觉和没有生命的物质。”Salingaros 博士总结说，Alexander 的观点是强烈的道家观点，而他的观点是首先在《建筑的永恒之道》一书中发展起来的。

为技术人员写的易经

亚利桑那大学的 Joseph Boudreaux [CCIT] 说，《建筑的永恒之道》以及《建筑模式语言》两本书是关于解决任何领域中的问题的设计模式和结构的书。这两本书就像 Java 或者 C++的书一样，只是没有讨厌的代码语法。读着这两本书，就好像读着一本为技术人员写的易经或者是儒家—道家的著作一样。

混沌之死

在从道家的角度来诠释 Alexander 的观点之前，先来看一看《庄子・应帝王》中的一个寓言故事。（这个寓言故事被笔名为“瓜”的作者做了非常有意思的改写[HUNDUN]。）

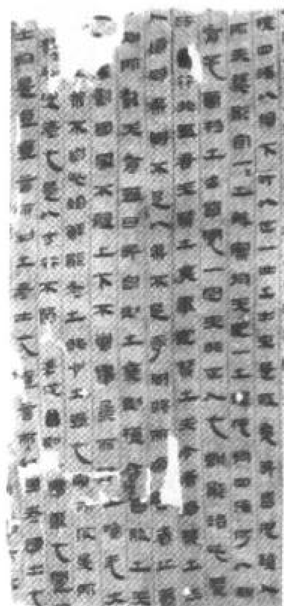
“南海之帝为倏，北海之帝为忽，中央之帝为混沌。倏与忽时相与遇于混沌之地，混沌待之甚善。倏与忽谋报混沌之德，曰：‘人皆有七窍，以视听食息，此独无有，尝试凿之。’日凿一窍，七日而混沌死。”

这个故事讲的是没有感觉器官的混沌，在得到感觉器官之后，就死掉了。混沌是在人的感觉之前存在的，在进入了人的感觉之后，混沌就不存在了。

老子说，“无名天地之始，有名万物之母”。无名，是无法定义的意思。人脑认识一个客观实体的过程，就是界定一个实体的过程，也就是要给它定出界限。道家认为，一个实体是无法与其环境分割开来的，因此，将一个实体局限到一个范围之中的过程，也就是改变它的过程。一个实体一旦被界定，也就不再是原始的那个实体了。

只有在一个实体被界定之后，才会在人脑中产生概念，这就是“名”。“名”就是死去的混沌。

大家知道，在经典力学中，波与粒子是完全不同的东西，波与粒子的属性是不能同时存在的。但是在量子物理学中，一个物理实体既具有波的属性，又具有粒子的属性。当实验探测的是波的性质时，微观粒子所显示出来的就是波的性质；相反，如果实验探测的是粒子的性质时，微观粒子所显示出来的就是粒子的性质。而实验做出的探测哪一种属性的决定完全可以是随机的、不预先设定的。这就表明，一个微观粒子本身必须同时具备波和粒子的性质。

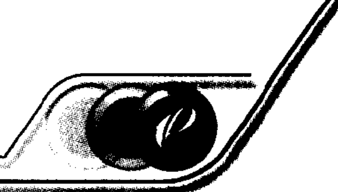


没有观察者的介入，一个被观察的实体是无法界定的，因此，老子说“无名天地之始”。庄子的“混沌”就是“无名”的人格化。

一个实体被界定的方式可以千变万化，因此同一个实体可以产生不同的“名”。当人们试图界定宇宙的本质时，就会发现一千条、一万条属性，这些都是宇宙被不同方式所界定的结果。人们所见的万物，都是宇宙的一部分，是人脑将宇宙局限到某一个区域中的结果。因此，道家说“有名万物之母”。

马王堆出土的《道德经》片段如右图所示。

在量子物理学中，对具有“波-粒二相性”的微观粒子进行探测的时候，实验就干扰了被观测的实体，使得后者呈现出可以



概念化的“名”，也就是波或者粒子的属性。任何的量子物理实验都是观察者对客观实体进行干涉的过程，所有的观测结果都是观察者对被观察者进行界定的结果。

再论无名的质（QWAN）

所谓“质”（又称“无名的质”，或者简写为 QWAN），所代表的概念非常接近于道家的“名”。

QWAN 是一个建筑物所具有的属性，当人脑试图认识它的时候，不可避免地会将它局限到不同的区域中。这个时候，这个无名的“质”就变成了一些可以定义的“质”，包括整体性（wholeness）、完备性（completeness）、舒适性（comfort）、和谐性（harmony）、可居住性（habitability）、持久性（durability）、开放性（openness）、可变性（variability）、可塑性（adaptability）等。

QWAN 是建筑设计依据的原始根据，而这些可以定义的“质”便是一切工程设计的开始。

微观粒子的“波–粒二相性”就是 QWAN 概念的物理存在。QWAN 就是老子的“无名”，或者庄子的“混沌”。

再论门

这一概念实际上非常接近于我国道家的“气”。《道德经》说，“万物负阴而抱阳，冲气以为和。”换言之，万物皆由阴阳组成，同时气又是阴与阳的中介和载体，气和阴阳是一个东西的两个侧面。气很接近于近代物理学的力场或物质波，而波与粒子则是一个物理实体的两种属性。

唐代的成玄英（约公元 601～690 年）在《庄子疏》中说：“气聚而有其形，气散而归于无形也”。换句话说，气以“形”体现出来，而“形”便是模式，模式的语言便是“形”的语言。

在粒子物理里面，微观亚原子粒子通过交换中介粒子产生相互作用，而这些中介粒子就成了道家的“气”的概念的物理实体。

再论道

《道德经》说：“道生一，一生二，二生三，三生万物。万物负阴而抱阳，冲气以为和。”这就是说，万物是由道而生，第一个实体就是内部蕴含有阴、阳两极的一个整体。这就等于否定了任何阴或者阳单独存在的可能。

与此类似，Alexander 在《建筑的永恒之道》一书的“空间的分化”中写道：“在自然界中，一个实体总是作为一个整体出生、发展的。”他认为一个建筑设计的过程，就像一个受精卵的发育过程一样，是一个逐步分化的过程，是从整体到部分的过程，而不是从部分到整体的过程。

由此可见，模式的哲学很接近于道家哲学，是形而上学，形而上学始于“形”之上。



希望大家在学习模式时，不要忘记模式（“形”）还有“而上之学”。

由于形而上学的相似性，中国人学习模式时应当尽量采用中国人固有的思维方式，因为我们的思维方式最适合于演绎模式理论。

建筑风水学与软件风水学

“气乘风则散，界水则止”，风水描述是只存在于建筑物及其环境所形成的整体之中，而不存在于独立的各个部分之中的一些属性。西方的建筑美学，主要注重在街区建筑的外在形式美，如对景、韵律、景观图案的几何构图等。中国风水美学，主要注重内在的“灵魂”美，其理论基础建立在河洛精蕴之中，易理太极之上。

风水强调建筑与环境的相互作用，认为建筑物与其环境是一个和谐的整体，这个整体会给居住在其中的人带来特定的体验。比如，一个花瓶的美，不仅仅存在于花瓶的形状及其表面装饰上，而且存在于花瓶所在的位置及其环境中。一个物体与其环境的相互作用是通过“气”产生的。如果这个相互作用是和谐的，那么“气”就以美的形式表现出来。

与此相类似，Alexander 在《建筑的永恒之道》一书中写道，一个花园之所以“有生命”，是由于植物、风、动物都处于完美的平衡之中的缘故。

读过 Alexander 的《建筑的永恒之道》的读者不难看出，Alexander 的 QWAN 实际上就是“风水”的别名。Alexander 的建筑的永恒之道，实际上就是“风水学”的一个“科学版本”。

风水在中国流行了几千年，在北方，大多与坟墓方位和活人的权势有关；在南方，大多与住宅的位置和财运有关。风水总是与迷信相关联，与功利相关联，使得风水在中国一直很难退去其神秘主义的外衣，变成科学的一部分。

Alexander 的著作可以提醒有中国文化背景的读者，注意“风水学”内在的合理性。如果能够成功地将迷信和功利的因素剥离，而重新以理性、科学（包括心理学）的方式塑造“风水学”，就可以大有作为。

对于一个软件设计工程师来说，一个软件系统也必定有某些属性仅仅属于其整体，而不存在于其某个部分的情况。换言之，读者可以思考，软件系统设计可否有“风水学”，这个“软件设计风水学”与现有的设计模式理论有何关系等。

1.5 软件的永恒之道

通过遵循 Alexander 的观点并且追溯到他的观点的道家起源，可以将他的理论“投射”到软件设计中。

三论无名的质（QWAN）

首先，QWAN 意味着一个软件的内在属性不仅仅存在于这个软件之中，而且存在于这个软件与其他软件的相互作用之中，存在于这个软件与计算机外部的世界，特别是用户的相互作用之中。



这个最重要的属性之所以是“无名”的，是因为它是映射到人脑之前的属性。在这个属性被映射到人脑中之后，就已经经历了用各种方式局限到各种不同范围之内的过程。这种过程导致这个无名的质变成一大堆有名的“质”。在本书后面，读者会读到这些有名的“质”包括软件性能要求的可变性、软件的可用性，以及系统的可扩展性、灵活性和可插入性等。

如果读者觉得这个概念仍然难以理解的话，不妨干脆认为一个软件的 QWAN 就是一个软件的“风水”。

三论门

对于 Java 语言设计师来说，“气”就是 Java 源代码，这些源代码汇聚而形成“形”，而其中的一些“形”反复出现，构成对一些典型问题的典型解决方案，这就是模式。人们给这些“形”定义一个可以区别的名字，以便能够相互沟通。人们研究这些“形”，从而可以积累经验，以便能够举一反三地处理类似的问题。

“理寓于气，气囿于形”，一个模式中会有几个角色存在，这些角色本身也是“气”汇聚而成的，并且通过“气”相互发生作用。根据“形”的尺度模式可以分成：“代码模式”，也就是由 Java 语句组成的“形”；“设计模式”，也就是由 Java 类和对象组成的“形”；“架构模式”，也就是由大尺度的构件组成的“形”。

三论道

软件的设计必须从软件的整体属性，也就是 QWAN 出发，通过性能的分化，形成各个模块、各个用户界面、各个类，以及各个方法。在软件的性能要求发生变化的时候，QWAN 是不会变化的。一个软件如果能够保持 QWAN，就能保证在变化的世界中保持一个软件设计的稳定性，这就是软件设计的永恒之道。

QWAN 必须通过对“道”的追随，经过“门”而实现。对一个 Java 设计师而言，这就意味着要经过 Java 源代码来形成好的“形”，从而达到软件系统的 QWAN。

如果读者进一步阅读本书的话，就会发现，“开–闭”原则非常接近这样的“道”。它在不同条件下分化为几条不同的设计原则。这些设计原则都是“永恒之道”在不同方向上和不同层次上的体现。

设计模式包括本书将要介绍的 26 个设计模式，都是帮助设计师到达 QWAN 的途径。

对哲学的呼唤

建筑物的建造会受到很多物理规律的限制，比如重力、材料强度等。而软件存在的世界是一个虚拟的世界，是一个没有重力、没有空气阻力、没有摩擦力的世界。这使得软件工程学与一般的工程学有很大的区别，而其中重要的一点区别是人们可以在一个软件系统中，以很少的成本做各种实验，这将何其自由。

软件是人类思维活动的直接延伸，软件将人类心理学、人类信仰、科学、艺术综合在

一起，这又与中国古代的思考方式何其相近。

如果读者阅读本书后面所讲解的一系列软件系统设计原则的话，就会发现这些原则与中国古代思想家在完全不同的领域、对完全不同的问题提出的观点何其相似。

正如量子物理学在上个世纪初处在科学哲学的前夜一样，毫无疑问，软件工程学现在正处在产生自己的科学哲学的前夜。具有中国文化传统的人们在这个重要关头享有很大的优势，因为现有的这些设计原则和模式的研究与中国的道家思想的联系是何其紧密。

希望“软件工程哲学”的太阳能够从东方的地平线上升起。

1.6 模式的要素

Alexander 说，“我们定义的每一个模式都必须依照一定的法则构造出来，以便能够建立起环境，列出此环境里的力，以及一个能够平衡这些力的位形”。并且 Alexander 建议使用图形来说明所研究的模式。

描述模式需要一定的格式。描述模式的格式有很多，在 Alexander 的著作中使用的格式就叫做 Alexander 格式。在“四人帮”的著作中使用的格式就叫做 GoF 格式。在 GoF 格式中，“四人帮”沿用了一些 Alexander 格式中的段落标题，这些标题常常叫做正则格式。尽管在不同格式中，正则格式的细节有所不同，一般来说，大家均同意模式应当包含以下这些要素：

- 名字（Name）：一个模式必须有一个有意义的、简短而准确的名字。一个好的名字可以使对模式的抽象讨论变得容易。有时一个模式同时有多于一个的名字，这时候这些名字就应当作为别名列出。
- 问题（Problem）：每一个模式必须有一个能够描述它的用意的问题，以便能够说明此模式在给定的环境和力中要达到的目标和效果。经常发生的是，力彼此抵消，并且促进或者阻止模式达到它所希望达到的效果。
- 环境或初始环境（Context 或 Initial Context）：模式的问题和模式的解答出现所需要的前提条件。环境说明模式的适用范围，环境也是模式应用之前的起始位形。
- 力（Forces）：力给出与模式相关的力和约束，它们怎样彼此相互作用，以及它们对实现目标所起的反作用，比如一个模式为所得到的好处付出的代价等。要比较好地描述一个模式，就必须完整地给出所有对模式有影响的力。
- 解答（Solution）：解答相当于一个生产产品的指令，它可能包括相片、图表、文字，用于确定模式的结构、所涉及的角色以及角色之间的协作，解答要显示出问题是怎样得到解答的等。解答不仅要给出静态的结构，而且要给出动态的行为。其中静态的结构描述此模式的形式和组织，而动态的行为使得模式变成“活”的。模式的解答所描述的正好可以作为实现模式的指南。
- 举例（Examples）：可以使用一个或多个示意性的应用来说明特定的真实环境，以及模式是怎样应用到环境上、改变环境并且给出当模式结束时的末态环境的。例子可以有助于读者理解模式的使用方法和适用性。每一个例子均可以附带有一个实现的样本，说明解答是怎样给出的。从熟知的系统里取出来的，有视觉效果

的或以比喻方式表达的例子可以更易于使用者理解。

- 末态环境（Resulting Context）：模式应用到系统之后的状态或位形。末态环境包括模式带来的好的和坏的结果，以及新状态中含有的其他问题和可能设计的其他有关系的模式。末态环境也就是模式的末态条件和可能有的副作用。末态环境有时也叫做力的解决，因为它描述的是什么力得到解决，什么力没有得到解决，以及什么模式可以应用到特定的系统上。很好地描述末态环境可以帮助比较末态环境与起始环境的区别和联系。每一个模式的每一次应用都是实现一个较大的任务的一小步。
- 推理（Rationale）：推理解释本模式的步骤、规则，以及此模式作为一个整体是如何以特定的方式解决模式的力的，解释本模式的力和约束是怎样协同合作的。它可以告诉设计师模式是怎样工作的，为什么可以工作，以及为什么此模式是好的。模式的解答可以描述模式外部的可见的结构和行为，而推理可以给出模式在系统表层以下的深层结构和关键机制的洞察力。
- 其他有关模式（Related Patterns）：这一要素应当描述在现有的系统中这个模式与其他的模式的静态的和动态的关系。相关的模式经常带有相同的一些力。它们的初始环境和末态环境经常是相容的。这些模式有可能是本模式的前任模式，也就是说，应用了这些模式可以给出本模式的初始环境；也有可能是本模式的继任模式，也就是说，本模式的应用给出这些模式的初始环境。这些模式还有可能是本模式的替换模式，也就是说，它们在不同的力和约束下，可以给出相同问题的不同解答；也有可能是本模式的相互依赖的模式，可以或必须和本模式同时使用。
- 已知的应用（Known Uses）：在已经有的系统中模式出现和应用的例子，都可以作为已知的应用。这有助于证明此模式确实是对一个重复发生的问题的可行的解答。已知的应用经常可以成为教学用的教材。

任何对模式的讲解或论述必须依据一定的格式，正如 Alexander 选用 Alexander 格式，GoF 选用 GoF 格式一样。但是，所有的模式都应当包括模式的所有要素，本书的格式也不例外。

1.7　本书讲解模式的格式

与所有其他讨论和讲解模式的书籍一样，必须选择一个适合于本书的讲解格式。本书是一本面向初学者的书，是一本通俗的教程；同时本书也为想要进一步研究模式理论的读者提供向深一层发展的切入点。因此，本书采取了以下的讲解格式：

- 模式的介绍：包括模式的名字和别名、摘要，以及模式的用意；
- 模式的结构：包括模式的示意性 UML 图、示意性的 Java 源代码、模式所涉及的角色，以及这些角色如何相互作用；
- 模式的长处和短处：对模式使用之前和之后的环境加以比较，可以说明设计师使用模式能够达到什么样的目标；
- 模式应当在什么情况下使用：也就是模式可以在什么情况下，针对什么样的问题

使用，可以改善什么样的不良设计；

- 关于模式实现的讨论：说明模式在 Java 语言中实现的时候可能遇到的问题，以及相应的解决方案；
- 举例：包括在读者熟知的 Java 语言 API 中的应用，以及其他常见的应用举例；
- 相关的模式：与相似的模式的比较，以及与协同使用的模式的关系。

另外，每一节都设有多个问答题，所有的问答题都有完整的答案。希望可以满足把本书当做教材的要求。

参考文献

[PLOP95] James O. Coplien and Douglas C. Schmidt (ed.). Pattern Languages of Program Design. Addison-Wesley，1995

[PLOP96] John M. Vlissides，James O. Coplien and Norman L. Kerth (ed.). Pattern Languages of Program Design 2. Addison-Wesley，1996

[PLOP98] Robert Martin，Dirk Riehle，Frank Buschmann (ed.). Pattern Languages of Program Design 3. published by Addison Wesley，1998

[PLOP99] Neil Harrison，Brian Foote，Hand Rohnert(ed.). Pattern Languages of Program Design 4. published by Addison Wesley，1998

[ALEX64] Notes on the Synthesis of Form. Harvard University Press，1964

[ALEX75]The Oregon Experiment. Oxford University Press，1975（此书中译本[ALEX75Z]）

[ALEX77] A Pattern Language: Towns，Buildings，Construction. Oxford University Press，1977（此书中译本[ALEX77Z]）

[ALEX79] The Timeless Way of Building. Oxford University Press，1979（此书中译本[ALEX79Z]）

[ALEX75Z]（美）C.亚历山大 M.西尔佛斯坦 S.安吉尔 石川新 D.阿布拉姆斯.《俄勒冈实验》. 赵冰，刘小虎译. 知识产权出版社，2002-2-1

[ALEX77Z]（美）C.亚历山大 S.伊希卡娃 M.西尔佛斯坦 M.雅各布逊 Z.菲克斯达尔一金 S.安吉尔.《建筑模式语言》（上、下册）. 王听度 周序鸣译. 知识产权出版社，2002-2-1

[ALEX79Z]（美）C.亚历山大.《建筑的永恒之道》. 赵冰译. 知识产权出版社，2002-2-1

[APPLETON] Brad Appleton . Patterns and Software: Essential Concepts and Terminology.（http://www.enteract.com/~bradapp/）

[SALIN] Nikos A. Salingaros. Some Notes on Christopher Alexander. http://www.math.utsa.edu/sphere/salingar/Chris.text.html

[CCIT] Joseph Boudreaux. Whatcha' readin'? Computing and Communications News. The University of Arizona，Center for Computing and Information Technology，March 2002

[HUNDUN] 瓜，混沌——创世神话的瓜式重构. 橄榄树文学月刊，2001 年 4 月刊

第 2 章　统一建模语言 UML 简介

本书在后面会使用 UML（统一建模语言，Unified Modeling Language）说明模式的结构和各个示范系统、例题的结构。由于本书在后面使用到的 UML 较为简单，且大多集中于类图、状态图和时序图，因此本书只针对这些类型的 UML 图做简单的介绍。没有接触过 UML 的读者，只要用心阅读本节的介绍，就能看懂后面的讲述。读者如果有兴趣进一步学习 UML，可以参考专门的 UML 教材。

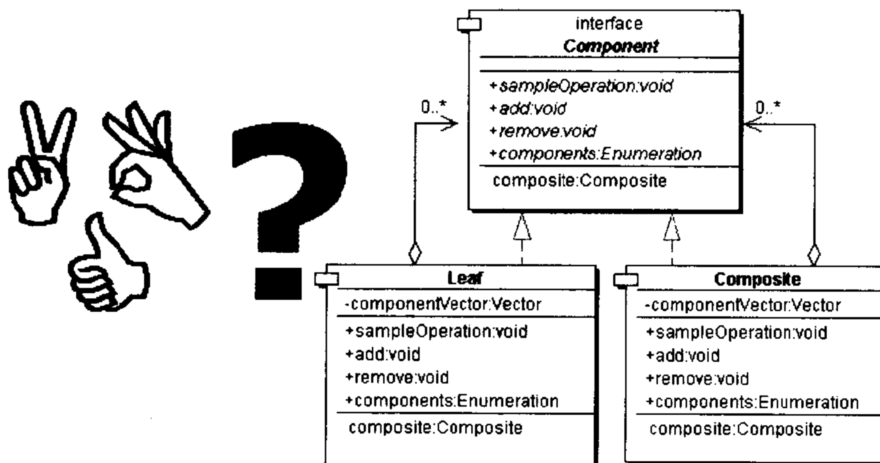
如果读者在阅读期间，需要临时查找 UML 图标及其在 Java 语言中的实现的话，可以使用本书的“附录 B　UML 图标及其 Java 实现一览表”。

2.1　建造世界贸易中心

一个正在学习计算机的朋友在一个门窗制造工厂里工作。工厂的技术人员大多都没有工程方面的训练，因此他们都是按照自己的方式进行设计和建造的。由于管理人员要求每一个设计都要有设计图纸，所以他们在按照自己的方式设计好门窗之后，甚至是造好门窗之后，才画一个设计图敷衍上级管理人员。他们认为，这些设计图纸都是一些形式，是浪费时间，他们按照自己的方法足以把工作做好。

有一次，这个朋友和笔者远远地观看纽约世界贸易中心原址正在清理建筑物残骸。他指着其中的门窗残骸说，他可以设计出比那些门窗更加漂亮的门窗。“连设计图纸都不用吗？”作者不禁想问。

设计图纸是设计的语言，是不同的工程设计人员之间、设计人员与生产人员之间沟通的语言。在一个生产作坊里面，人们沟通可以使用手语，打着半哑谜。在一个现代化的工程里面，人们要相互沟通和协调合作，就必须使用标准的工业化设计语言，如下图所示。



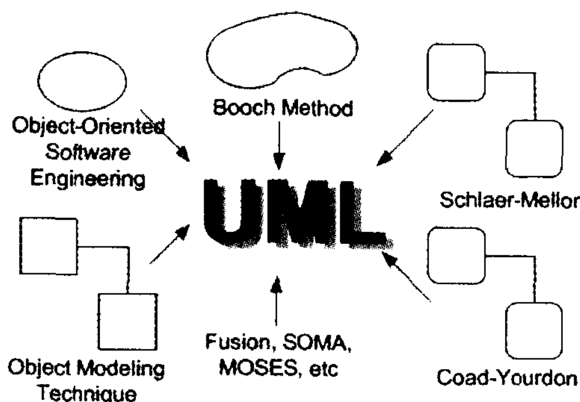
如同电子工程师有电路设计图，机械工程师有机械设计图，建筑工程师有建筑设计图一样，软件工程师如今有了 UML 图，UML 是当今软件设计的标准图标式设计语言。

在国内，UML 的使用正在逐渐得到重视，但是仍然没有成为设计人员交流的语言。在很多开发项目里，人们在系统设计过程中使用的仍然是“手语”和“半哑谜”一样的沟通方式，UML 仅仅被当做点缀。就如同那个制造门窗的朋友一样，人们认为 UML 仅仅是一种形式，不使用 UML 或者任何一种标准设计语言，他们“仍然可以建造出世界一流的软件产品”。

显然，这和不使用标准设计图纸，就要建造出世界贸易中心的“雄心壮志”在本质上不是一样的吗？

2.2 什么是 UML

统一建模语言 UML 是 OMG（Object Management Group）在 1997 年发表的图标式软件设计语言，它综合了很多种当时已经存在的面向对象的建模语言、方法和过程，如下图所示。



对一个软件系统而言，UML 语言具有以下的重要功能[BOOCH99]：可视化（Visualizing）功能、说明（Specifying）功能、建造（Constructing）功能和建文档（Documenting）功能。

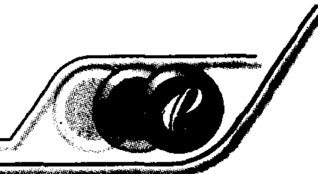
可视化功能

可视化可以促进对问题的理解和解决，并且方便熟悉 UML 的设计师彼此交流和沟通。

可以较容易地发现设计草图中可能的逻辑错误，保证最后完成的软件确实能按照要求运行，避免和减少意外发生。

说明功能

对一个系统的说明应当通过一种通用的、精确的、没有歧义的通信机制进行，显然 UML



的特性使得 UML 很适合于这种说明工作。

系统的整体设计可以指导软件的开发过程。由于重要的决定均可以在开始写代码之前就做出，因此可以减少低质量的代码，进一步降低开发成本。

建造功能

UML 有它自己的语法规则，这使得人们可以使用建模工具软件对一个系统设计模型加以解释，并将设计模型映射到一种计算机语言（如 Java）上。这也就是说，使用一种建模工具可以大大加快建模和系统设计的过程。

通过 UML 可以看到总体的图像，这样一来，可以均衡调配系统所消耗的计算机的资源，使系统更有效率。因为系统的设计首先完成，所以很容易就能发现可以复用的代码。代码能够高效率地实现复用，可以降低开发成本。

建文档功能

使用 UML 进行设计可以同时产生系统设计文档。

由于使用 UML 设计的软件系统在写出代码之前就有了专业化的设计和文档资料，所以程序员事先精确地知道他们的计划是什么。当需要修改一个已有的系统时，如果能找到那个系统的 UML 文档资料，则会节省学习时间，使修改工作事半功倍。这样可以降低维修成本。

如果在项目进行过程当中，有新的程序员参加项目的话，这些程序员可以借助 UML 图形文档资料很快熟悉开发中的系统。

2.3　UML 包括什么

UML 包括以下的图：

- 使用案例图（Use case diagrams)
- 类图（Class diagrams)
- 序列图（Sequence diagrams）
- 合作图（Collaboration diagrams）
- 状态图（Statechart diagrams）
- 活动图（Activity diagrams）
- 构件图（Component diagrams)
- 部署图（Deployment diagrams）

在所有的这些图里面，案例图、类图和序列图是最为有用的。

根据这些图的用意，可以将它们大体上划分为结构型图和行为型图两种。结构型图描述了系统的静态结构，在显示一个系统已有的类及它们之间的静态关系时最为有用。行为型图描述一个系统的动态性质，在显示系统的元素如何协作产生满足要求的系统行为方面

最为有用。

结构型图

如下表所示，其中给出了所有的 UML 结构图。当然，在所有的结构型图里面，**类图**是最为常用的。

图的名字	介绍
类图（Class Diagram）	类图描述一些类、包的静态结构和它们之间的静态关系
对象图（Object Diagram）	对象图给出一个系统中的对象的快照
构件图（Component Diagram）	描述可以部署的软件构件（比如 jar 文件，EJB 等）之间的静态关系
部署图（Deployment Diagram）	描述一个系统的拓扑结构

显然，要描述一个设计模式的静态结构，使用类图和对象图是很合适的。

行为型图

如下表所示，其中给出了所有的 UML 行为型图。当然，在所有的行为型图里面，**案例图和时序图**是最为常用的。

图的名字	介绍
使用案例图（Use Case Diagram）	使用案例图描述一系列的角色和使用案例及它们之间的关系。可以用来对一个系统的最基本的行为进行建模
活动图（Activity Diagram）	描述不同过程之间的动态接触。活动图是使用案例图所描述的行为的具体化
状态图（State Diagram）	描述一系列对象的内部状态及状态的变化和转移。注意一个类不能有两个不同的状态图
时序图（Sequence Diagram）	时序图是一种相互作用图，描述不同对象之间信息传递的时序
合作图（Collaboration Diagram）	合作图是一种相互作用图，描述发出信息、接收信息的一系列对象的组织结构

显然，要描述一个设计模式的行为特性，使用状态图和时序图就很合适。

只要有意义，所有类型的 UML 图都是可以混合在一起使用的。比如，一个对象图可以与一个类图同时出现在一个结构图中，一个构件图中可以有类图出现等（有的时候，一些 UML 建模工具软件不一定允许这样做）。

而在本书中，类图、状态图和序列图是最为常见的图。

应当指出的是，一个使用 UML 的系统设计，往往是从使用案例图开始的，而且一个设计应当是以使用案例驱动的。由于本书并不涉及到真实的商业需求，所涉及真实商业项目的部分，也仅仅限于技术观点上的讨论，因此在本书出现了案例图。

同样，由于本书并不涉及软件系统的部署问题，因此构建图和部署图在本书中用不到。

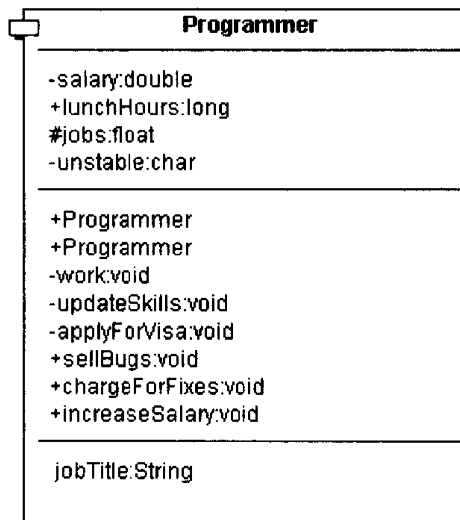
在下面，本书仅就将要使用到的类图、状态图和序列图做简单的介绍。

2.4 类　　图

类图（Class Diagram）是显示出类、接口以及它们之间的静态结构和关系的图。类图最基本的元素是类或接口。

描述类的类图

下图所示就是一个类。



在上面的类图中可以看出，表示类的框分成以下几层:

- 类名
- 属性清单
- 方法清单
- 性质清单

如果一个类有内部成员类，它的类图就会有五层。在类的类图中，除了类名层是不能省略、必须显示的以外，其他几层都是可以在 UML 图中省略的。

第一层是类名。类名如果是正体字，表明类是具体的（Concrete，即可以实例化的类），变量名如是斜体字，表明类是抽象的。 显然在图中给出了一个具体的类。类名是不能省略、必须显示的。

第二层是属性层。一个属性可以是 public、private 或 protected。一个属性的左面如果有一个加号（+），表示它是 public；左面如果有一个减号（-），表示它是 private；左面如果有一个井号（#），表示它是 protected。

第三层是方法层。一个方法的左面如果有一个加号（+），表示它是 public；左面如果有一个减号（-），表示它是 private；左面如果有一个井号（#），表示它是 protected。

在 aPublicStaticMethod()方法的下面有一道下划线，表明这是一个静态的方法。

一个方法有几个要素：方法的名字、方法的变量名与变量的数据类型，以及方法的数据类型。比如在下面的方法中，方法的名字是 aPublicMethod， 方法的变量有两个，分别是 param1 和 param2，其数据类型分别是 int 和 String，而方法的数据类型是 String，如代码清单 1 所示。

代码清单 1：方法举例

```
public String aPublicMethod(int param1,   String param2)
{
    return "return value = " + param1 + param2;
}
```

一个类的构造子也属于方法，但是构造子是特殊的、没有返还类型的方法。一个类的构造子不一定是 public，它也可以是 private 或 protected。一些水平尚好的程序员对这一点都认识不清：为什么构造子可以是 private 呢？详情请见本书"单例（Singleton）模式"一章的介绍。

第四层是性质层。性质是由一个属性即由一个内部变量，一个赋值函数（mutator）和一个取值函数（accessor）组成的结构。比如在上图的例子里，property1 是一个性质，其结构可由代码清单 2 看出。

代码清单 2：性质的源代码

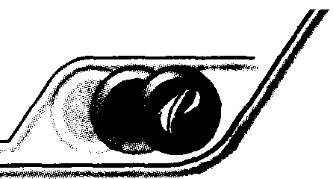
```
public String getJobTitle()
{
    return jobTitle;
}
public void setJobTitle(String jobTitle)
{
    this.jobTitle = jobTitle;
}
private String jobTitle;
```

在类的方框右上角，通常还分两行写出类的父类和所实现的接口。在后面读者会看到例子。

Programmer 类的源代码如代码清单 3 所示。

代码清单 3：Programmer 类的源代码

```
public class Programmer
{
    //私有属性
    private double salary;
    //公开属性
    public long   lunchHours;
    //保护属性
    protected float   jobs;
    //私有属性
    private char    unstable;
```



```
//私有属性
private String jobTitle;
//公开默认构造子
public Programmer()
{
     //default constructor
}
//公开构造子
public Programmer(String jobTitle)
{
     this.jobTitle = jobTitle;
}
//私有方法
private void work()
{
      System.out.printin("working...");
}

//私有方法
private void updateSkills()
{
      //there is nothing here
}

//私有方法
private void applyForVisa()
{
      //get Visa extended
}

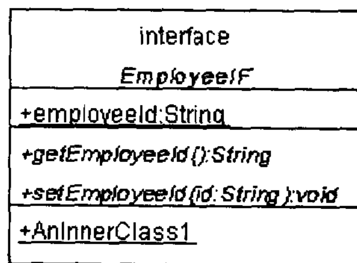
//公开方法
public void sellBugs(String customer)
{
     //sell bugs to customers
}

//公开方法
public void chargeForFixes(String customer)
{
     //charge customer for fixes
}

//公开方法
public void increaseSalary(double amount)
{
     this.salary += amount;
```

```
    }
    //公开方法
    public String getJobTitle()
    {
        return jobTitle;
    }
    //公开方法
    public void setJobTitle(String jobTitle)
    {
        this.jobTitle = jobTitle;
    }
}
```

描述接口的类图



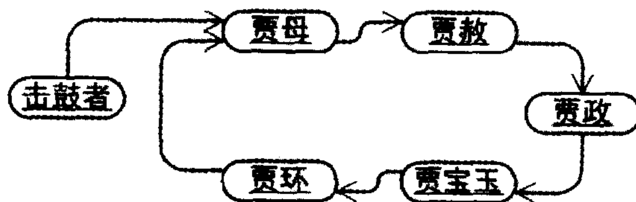
接口的类图与类的类图几乎一样，惟一的区别是接口的名有“interface”的字样。右图所示就是接口类图的一个例子。

一个接口不会有性质，可以有方法的声明、public 和 final 静态内部成员类。设计师可以选择只显示一个接口的各方法项示各属性项而不显示方法项等。

描述对象的类图

对象图是一种特殊的类图，它显示出的不是类而是类的实例。对象图在显示一小部分系统的结构关系时，特别是有递归关系的结构时非常有用。

在对象图中，每一个长方形都代表一个实例。实例的名字是有下划线的，这样可以区分一个长方形所代表的是类还是对象。只要图的含义是清楚的，那么对象的名字或者类的名字可以从图中省略，如下图所示。



在上面的对象图中，一共给出了六个对象，都是《红楼梦》中的人物，它们的名字都带有下划线。这些对象一个引用另一个，形成一个环形。对《红楼梦》中的故事有兴趣的读者，可以参阅本书的“责任链（Chain of Responsibility）模式”一章。

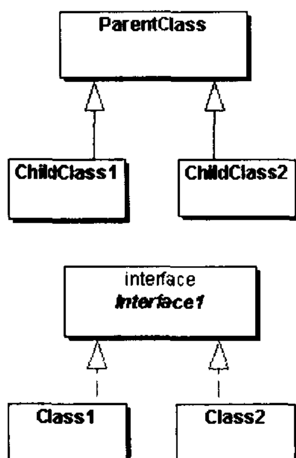
类图中的关系

在类与类之间，会有连线指明它们之间的关系。类和类、类和接口、接口和接口之间

可以建立以下几种关系：一般化关系、关联关系、聚合关系、合成关系和依赖关系，这几种关系都是静态的。

一般化关系

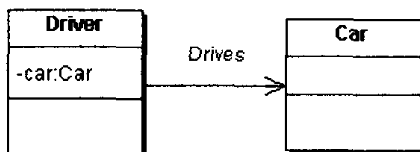
一般化（Generalization）关系表示类与类之间的继承关系，接口与接口之间的继承关系，或类对接口的实现关系。一般化的关系是从子类指向父类的，或从实现接口的类指向被实现的接口，与继承或实现的方向相反，如下图所示。



一般化关系在 Java 语言中可以直接翻译为关键字 extends 和 implements。前者描述类与类之间、接口与接口之间的一般化关系，后者描述与接口之间的一般化关系。

关联关系

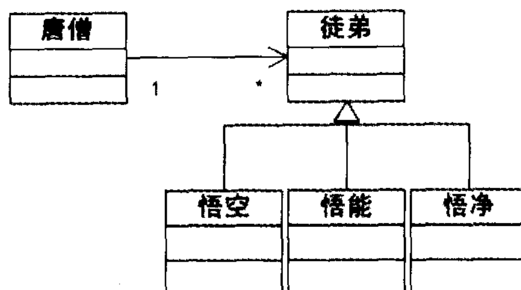
关联（Association）关系是类与类之间的联接，它使一个类知道另一个类的属性和方法。关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头。单向的关联有一个箭头，表示关联的方向，如下图所示。单向的关联更为普遍，通常不鼓励使用双向的关联。



在 Java 语言里，关联关系是使用实例变量实现的。比如在上面的 Driver 类中，就出现了一个类型为 Car 的实例变量，这个变量实现了这两个类之间的关联关系。每一个关联都有一个名字，在上面的例子里，关联的名字是 Drives。

每一个关联都有两个端点，每一个端点都可以有一个角色名，显示出关联的本质。一个关联可以有一个方向箭头，表明遍历或者查询的方向。

在每一个关联的端点，还可以有一个基数（Multiplicity），表明这一端的类可以有几个实例。比如，唐僧和他的徒弟形成一个关联关系，在这个关系里面，唐僧只能有一个，而徒弟可以有好几个，如下图所示。



常见的基数有下表所示的这些。

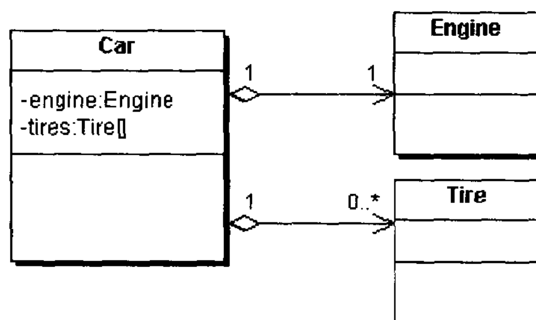
基　数	含　义
0..1	零个或者一个实例
0..*　或者*	对实例的数目没有限制（可以是 0）
1	只有一个实例
1..*	至少有一个实例

其中记号 $n..m$ 表明一个取值区间，也就是 n～m 个实例。

一个关联关系往往可以进一步确定为聚合关系或者合成关系。比如，唐僧与他的徒弟的关系就可以进一步确定为聚合关系。

聚合关系

聚合（Aggregation）关系是关联关系的一种，是强的关联关系。聚合是整体和个体之间的关系。例如，汽车类与引擎类、轮胎类，以及其他的零件类之间的关系便是整体和个体的关系。一个汽车对象是由一个引擎对象、四个轮胎对象组成的，如下图所示。



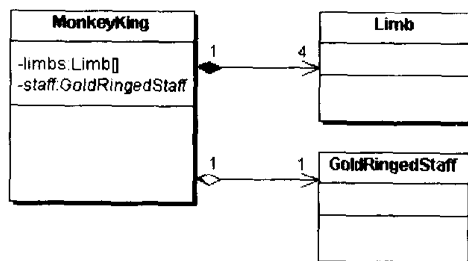
与关联关系一样，聚合关系也是通过实例变量实现的。以上面的例子为例，Car 类中应当有一个类型为 Engine 的属性和一个类型为 Tire 的数组属性。但是，关联关系所涉及的两个类是处在同一层次上的，而在聚合关系中，两个类是处在不平等的层次上的，一个代表整体，另一个代表部分。

关联与聚合仅仅从 Java 语法上是分辨不出的，需要考察所涉及的类之间的逻辑关系。如果读者不是很确定一个关系是不是聚合关系，可以将之设置为关联关系。

合成关系

合成（Composition）关系是关联关系的一种，是比聚合关系强的关系。它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期，合成关系是不能共享的。

代表整体的对象需要负责保持部分对象的存活，在一些情况下负责将代表部分的对象湮灭掉。代表整体的对象可以将代表部分的对象传递给另一个对象，由后者负责此对象的生命周期。换言之，代表部分的对象在每一个时刻只能与一个对象发生合成关系，由后者排他地负责其生命周期。聚合关系和合成关系的类图如下图所示。

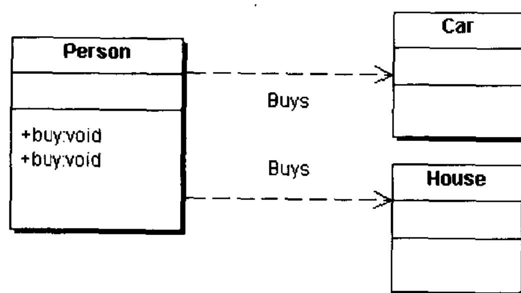


在上面的类图中，显示了美猴王（MonkeyKing）以及他的四肢（Limb）和他的金箍棒（GoldRingedStaff）之间的关系。可以看出，MonkeyKing 与 GoldRingedStaff 之间是聚合的关系；而 MonkeyKing 与 Limb 之间的关系要比前者更强，是合成关系，因为美猴王的四肢完全由美猴王自己负责，并且不能共享。

如果读者不能确定一个关系是不是合成关系，可以将之设置为聚合关系，甚至关联关系。

依赖关系

依赖（Dependency）也是类与类之间的连接，依赖总是单向的。依赖关系表示一个类依赖于另一个类的定义。一个人（Person）可以买车（Car）和房子（House），Person 类依赖于 Car 类和 House 类，如下图所示。



在上面的例子里，Person 类依赖于 Car 类和 House 类的定义，因为 Person 类引用了 Car 和 House。与关联关系不同的是，Person 类里并没有 Car 和 House 类型的属性，Car 和 House 的实例是以参量的方式传入到 buy()方法中去的。其源代码如代码清单 4 所示。

代码清单 4：Person 类的源代码（片断）

```
public class Person
{
```

```
        public void buy(Car car)
        {
            ....
        }
        public void buy(House house)
        {
            ....
        }
    }
```

一般而言，依赖关系在 Java 语言中体现为局域变量、方法的参量，以及对静态方法的调用。换言之，一个类 A 的某一个局域变量的类型是另一个类 B，那么类 A 就依赖于类 B。如果一个方法的参量是另一个类 B 的实例，那么这个方法所在的类 A 依赖于类 B。如果一个类 A 调用另一个类 B 的静态方法，那么类 A 依赖于类 B。

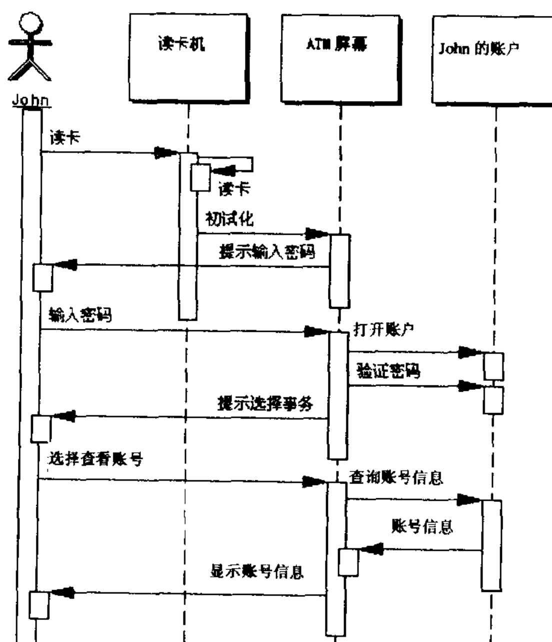
如果类 B 出现在类 A 的实例变量中，那么类 A 与类 B 的关系就超越了依赖关系，而变成了某一种关联关系。

每一个依赖关系都可以有一个名字。在上面的例子里，两个依赖关系的名字都是 Buys。

一般而言，每一个类图都应当有类、关联关系、基数。而关联关系的方向和关系中的角色是可选的，设计师可以在需要的时候加以强调。

2.5 时 序 图

时序图有时又叫做序列图、活动序列图。作为交互图的一种，序列交互图按照时间顺序从上往下显示每个使用案例。下图所示的例子显示了一个银行客户在 ATM 机器上查阅账号信息的时序图。



在一个时序图中，垂直的虚线叫做生命线，它代表一个对象存在的时间。每一个箭头都是一个调用，这个箭头从调用者对象连接到接收者对象的生命线上的激活条（Activation Bar）上。每一个激活条代表调用所持续的时间。

在上面的时序图中，John 向“读卡机”发出“读卡”调用，然后“读卡机”向自身发出同样的调用，然后将自身“初始化”。从这以后，“读卡机”向“ATM 屏幕”发出“提示输入密码”的调用，而后者则向 John 发出调用，要求输入密码。

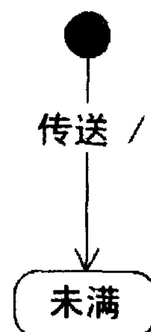
当 John 输入密码完毕后，“ATM 屏幕”向“John 的账号”发出“打开账号”和“验证密码”的命令。成功以后，“ATM 屏幕”又显示可以选择的事务给 John。

这时候，John 选择查看账号，这一调用是由 John 发给“ATM 屏幕”的，后者将这个指令转给“John 的账号”，然后执行这个指令，将账号信息传递给“ATM 屏幕”，而“ATM 屏幕”则向 John 显示出账号的信息。

2.6　状　态　图

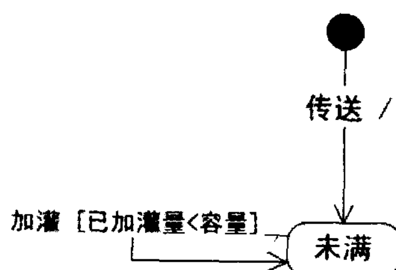
状态图（State Diagram）又称做状态转换图（State Transition Diagram）。状态图的基本想法是定义一个具有有限个内部状态的机器，因此状态图又称做有限状态机。对象被外界的事件激发，从而从一个状态转换到另一个状态。

下面考虑汽水装瓶机上的汽水瓶子的例子。对象是瓶子，装瓶机器是外部环境。首先，一个空的汽水瓶子在传送带上被送到汽水装瓶机龙头的下面，这时瓶子是空的。如右图所示。



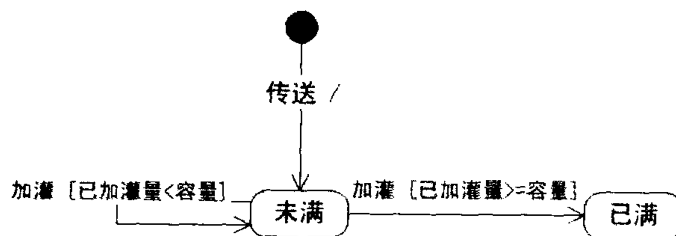
图中的黑点表示起始状态，方框表示瓶子的状态，可以看出瓶子没有满，处在“未满”的状态。从起始状态到未满状态的、有箭头的连线表示状态的过渡。过渡连线的标签通常分为两部分由一个斜线分开，斜线的第一部分是引起状态过渡的事件，第二部分是事件发生所引起的操作。本图中，只有第一部分，没有第二部分，因为瓶子并没有动作。

瓶子被加灌汽水，此时加灌条件是“已加灌量 < 容量”，如下图所示。在图中，未满状态有反身过渡连线，表示瓶子被加灌汽水是一个持续不断的过程，在“加灌”事件发生时，对象会过渡回当前状态。方括号中的事件是发生的保护条件，在这里是“已加灌量 < 容量”，它是保证过渡关系发生的条件。

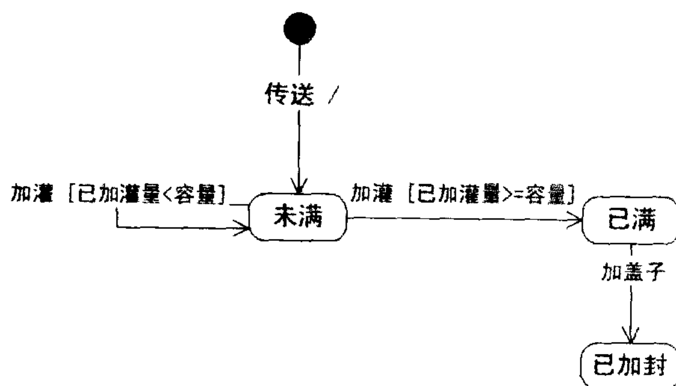


从瓶子被加灌汽水到瓶子达到“已满”状态为止，如下图所示。由图中可以看出，瓶

子从“未满”状态过渡到“已满”状态的事件仍然是加灌事件，只是条件不同，变成了“已加灌量 >= 容量”。

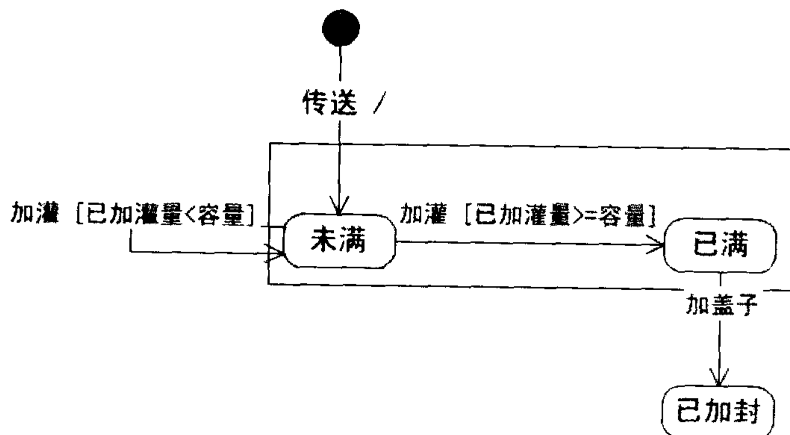


已被加满的瓶子将被盖上盖子。这里的事件是“加盖子”，瓶子过渡到“已加盖”状态，如下图所示。



显然，只有处在“已满”状态的瓶子可以被加盖子，处于“未满”状态的瓶子不会被加盖子。只有未满的瓶子会收到加灌汽水的事件，而已满的瓶子不会收到加灌汽水的事件。状态图可以让读者清楚地看到什么事件会发生到对象身上，以及发生的事件对对象会有什么样的效果。

状态可以嵌套。一个状态中可以有一些别的状态。大的状态叫做父状态或者超状态，小的状态叫做子状态。仍然以加灌汽水瓶为例，可以划分出一个超状态，它包括两个子状态，如下图所示。





状态图还有另外两个常用的图标，即历史状态图标和终态图标，如下图所示。





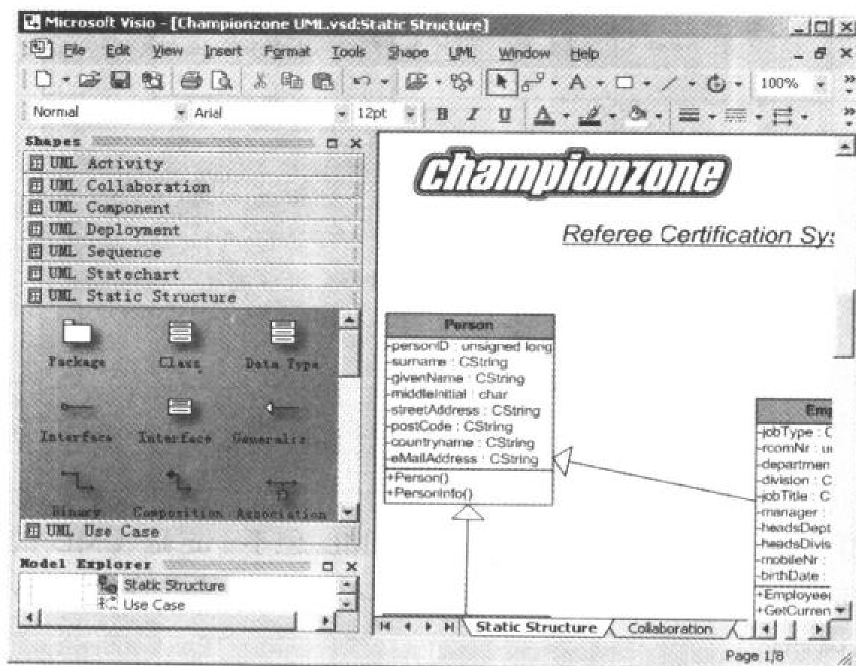
历史状态记住系统离开时的状态，终态图标是与初态图标相反的图标，代表对象湮灭的状态。

2.7　UML 及建模的工具

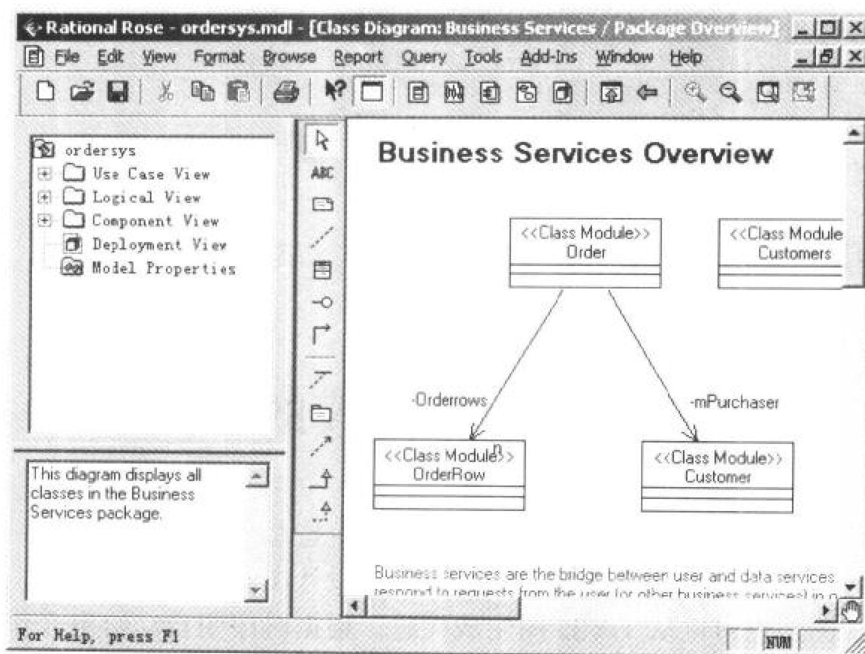
有很多工具可以帮助用户画出 UML 图。其中有些是纯粹的绘图工具，另外的一些则是有代码生成功能的 OO 设计工具。一个好的 OO 设计工具甚至可以双向工作，既可以从代码生成 UML 图，也可以从 UML 图生成代码；根据时序图自动给出合作交互图，或者根据合作交互图自动给出序列交互视图等。这样的工具包括如下内容：

- Rational Rose　www.rational.com
- Together　www.togethersoft.com
- Microsoft Visio　www.microsoft.com
- Visual UML　www.visualobjectmodelers.com
- Model Bridge　www.metaintegration.net
- MicroGold WithClass　www.microgold.com
- QuickUML　www.excelsoftware.com

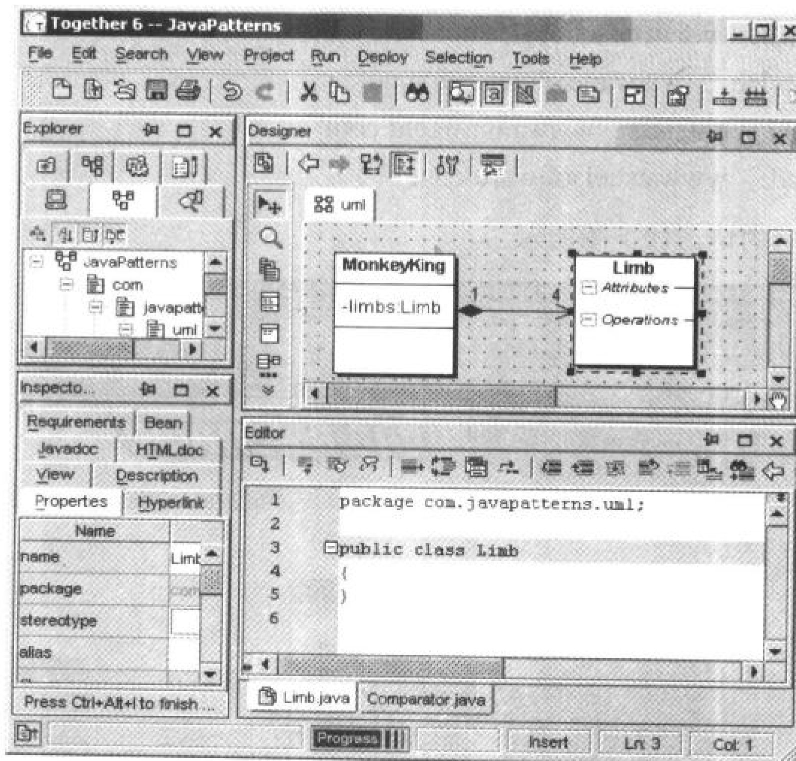
Visio 在运行时的情形如下图所示。



Rational Rose 工作时的情形如下图所示。



在以上列出的这些工具里面，Together 较为特殊。它把 Java IDE 和 OO 设计建模工具合二为一，用户可以同时看到 UML 图和 Java 源代码，修改 UML 图会使得源代码得到即时修改。反过来，修改 Java 源代码也会使 UML 图同时得到相应的修改，如下图所示。



在一些参考文献里，读者可以看到类似于 UML 但不一定是 UML 的图标，比如在[GOF95]里使用的是较早的 Booch 图标，而在[GRAND98]里使用的却是 UML 图标，本书通篇使用 UML 图标。

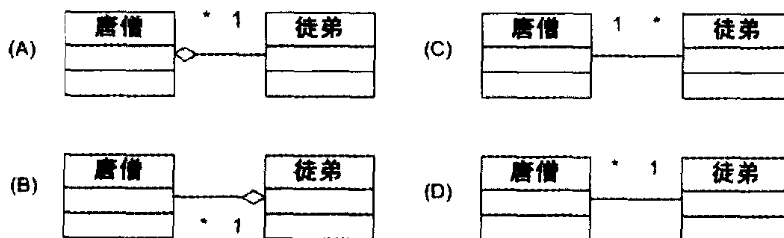
本书的 UML 图标大部分是使用 Together 和微软的 Visio 生成的，对 Together 感兴趣



的读者可以从 www.togethersoft.com 网站得到免费的试用版本。

问答题

1. 下面所示的这些类图试图描述"唐僧有几个徒弟"这一事实，请问哪一个是对的？



2. 下面关于状态图的描述里哪一个是对的？
   A. 状态图中的所有动作都连接到状态的转移上。
   B. 一个事件可能不改变一个对象的状态。
   C. 一旦一个对象离开了一个状态，那么它就不能回到那个状态。
   D. 从一个状态出来的两个状态转移可以交叉，或者说被同一个事件所激发。

问答题答案

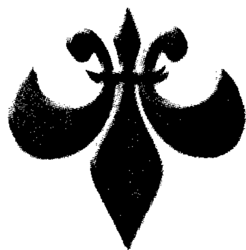
1. C. 是正确答案。
   A. 是错误的，因为这个图中基数的设定是错误的。
   B. 是错误的，因为这个图中聚合关联的方向是错误的。
   D. 是错误的，因为这个图中基数的设定是错误的。
2. B. 是正确答案，因为造成反身转移的动作不改变对象的状态。
   A. 是错误的，所有的动作都是写在状态里面的。
   C. 是错误的，一个对象可以离开了一个状态后，马上或在将来重新回到原有的状态。
   D. 是错误的，转移是不能交叉的。一个事件不能激发一个以上的转移。

参考文献

[BOOCH99] Grady Booch, James Rumbaugh and Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999

[BOOCH98] Grady Booch. The Visual Modeling of Software Architecture for the Enterprise. Rose Architect, 1998

[KIRK02] Kirk Knoernschild. Java Design – Objects, UML, and Process. Addison-Wesley, 2002



第二部分　面向对象的设计原则

如何同时提高一个软件系统的可维护性（Maintainability）和可复用性（Reuseability）是面向对象的设计要解决的核心问题。

在目前的软件界，学习和应用设计模式正在形成风气。的确，通过学习和应用设计模式，可以更加深入地理解面向对象的设计理念，从而帮助设计师改善自己的系统设计。但是，设计模式并不能够提供具有普遍性的设计指导原则。在读者经过一段时间的对设计模式的学习和使用之后，就会觉得这些孤立的设计模式的背后应当还有一些更为深层的、更具有普遍性的、共同的思想原则。

诸如“开–闭”原则，这样的面向对象的设计原则就是这些在设计模式中不断地显现出来的共同思想原则，它们是隐藏在设计模式背后的、比设计模式本身更加基本和单纯的设计思想。

在本书后面几章将要讨论的这些 OO 设计原则是提高软件系统的可维护性和可复用性的指导性原则。在有现成的设计模式的地方，设计模式就是这些设计原则在具体问题中的体现；在没有现成的设计模式的地方，这些设计原则也一样适用，同样可以对系统设计发挥指导作用，并对新模式的研究提供向导。

第 3 章　软件的可维护性与可复用性

通常认为，一个易于维护的系统，就是复用率较高的系统；而一个复用性较好的系统，就是一个易于维护的系统。但是实际上，可维护性和可复用性是两个独立的目标，就像两只奔跑的兔子一样，并不总是方向一致的。

对于面向对象的软件系统设计来说，在支持可维护性（Maintainability）的同时，提高系统的可复用性（Reuseability）是一个核心的问题。

3.1　软件系统的可维护性

一个在软件行业工作了二十几年的同事告诉我，一个软件开发只需要半年，维护则需要很多年。文献[WR00]则指出，西方发达国家的一个软件项目在其生命周期内，花在维护上面的钱，是花在原始开发上面的钱的两倍。

软件的维护

一件家用电器，比如电视机、放像机等，常常是买下后，一用就是好几年，在其生命周期之中，几乎没有什么维修费用可谈。人们在购买家用电器的时候，很少购买额外的维修保险，这表明轻视维护的思想根深蒂固。

一个家用电器的维护，只是保持或者恢复电器的某种操作性能所需要的时间和资源。一个软件的维护则不同，它不仅包括清除错误和缺陷，而且还要包括对已有性能的扩充，以满足新的设计要求。

家用电器的使用者不可能对一个黑白电视机进行改造，将它变成彩色电视，或者将一个小电视改造成为一个大电视。软件则不同，一个维护中的软件是一个不断再生的软件，就像一个“不断带来新礼物的礼物”。

一个软件业者必须认识到，软件的维护就是软件的再生。一个好的软件设计，必须能够允许新的设计要求以较为容易和平稳的方式加入到已有的系统中去，从而使这个系统能够不断地焕发出青春。

一个可维护性较好的系统，应当允许维护工作能够以容易、准确、安全和经济的形式进行。但是现有的软件系统设计往往不具有这样的特性。

一个典型的软件生命周期

如果读者已经在软件行业中工作过三年以上，也就是说，已经经历过至少一个软件生命周期，那么读者一定会同意下面对一个典型的软件项目的生命周期的描述。

一个软件项目开始了。系统设计师们拿到了系统的设计要求，这个设计要求有可能是一叠非常正规的文件，上面盖着某客户的合同专用章，也有可能是几页草稿纸和一个口头协议。无论是哪一种情况，现在系统设计师们开始进行系统设计了。

这个系统毫无疑问将是一个优美的系统。系统的美，首先存在于设计者们的头脑之中，然后存在于设计图纸之上，然后变成一个原型系统，最后变成一个真实的、有血有肉、可以交付客户使用的成品。

设计师喜欢看着它运行，程序员喜欢它的每一个功能键和图形视窗，而客户终于看到花出去的钱变成了现实。这个时候，这个系统就像一个美少女一样，毫无疑问是纯净的、优美的、动人心弦的。

但是不久事情就开始发生了变化。客户看了运行中的系统，又提出了一些“小小的”修改要求，这些要求都是客户在提出系统设计要求的时候遗忘的一些“小”问题。设计师讨论过之后，给出了一些设计上面的修改。由于这些修改与现有系统的设计并不一定相容，所以设计师们不得不采取一些权宜之计，使得这些修改看上去就像一个美丽少女脸上的青春痘。但是好像所有的软件都不能避免这些“青春痘”似的东西的出现，所以一切还不是很糟。

可是事情并没有就此结束，这些小的痘痘越来越多，而且其中有一些变成了系统的肿瘤。随着时间的流逝，这些肿瘤变成了系统中最主要的组成部分，当初的美少女变成了一堆丑陋不堪、日渐“腐烂”的代码，以至于没有人愿意去维护它。人们把它叫做 legacy 系统，而不得不去维护这个 legacy 系统的人们，会天天诅咒设计这个系统的设计师：“是谁设计出这么丑陋的一个东西？”

经过一段漫长的时间之后，“苦难”终于要结束了！在抱怨了软件系统的维护之昂贵之后，客户决定投入一笔资金，一个新的系统将被设计出来，而这个新系统将具有这个正在死去的系统的所有功能（当然还包括一些新的功能），并且取代这个已经“腐烂”的系统。

请读者想一想，这个新系统会面临什么样的命运呢？

设计师的辩解

面对着人们的抱怨，系统设计师都有一套一模一样的辩解：用户要求的变化无常，使得系统设计无法跟上如此快速的变化。如果维护设计师与原始设计师不是同一组人员的话，就还会有很多与技术无关的辩解出来。

按照这样的逻辑，一个系统的原始设计不可能预测系统的性能要求会发生什么样的变化，这就导致一个系统的设计无法与新的性能要求相容。这样一来，即便新的性能可以添加到系统中去，但是却不得不以某种破坏原始设计意图和设计框架的方式加入进去。

在很多情况下，一个系统的维护设计师并不是原始设计师。这样一来，这个维护设计师并不熟悉原始设计师的设计意图，而系统的设计也不会帮助维护设计师理解这些。因此，即便原始设计的意图和框架可以容纳新的性能，维护设计师也仍然可能以某种破坏原始设计意图和设计框架的方式将新性能加入进去。

由于这些改动是以积累的方式进行的，因此，维护设计师无法形成自己的设计意图和

新的设计框架，所以这些破坏都不会带来新的意图和框架的建设，而只能是一些东拼西凑的权宜之计。

这样的破坏式增强功能越来越多，以至于最后原始的设计意图和设计框架已经彻底被这些没有总体考虑和固定规律的东西取代，系统就这样“腐烂”了。

一个应当问的问题就是，软件系统的性能要求都是会变化的，难道变化的要求一定会导致系统“腐烂”吗？为什么一个系统的设计不可以为日后的变化留出足够的空间呢？

真正的原因

Rober C. Martin 在[MARTIN00][MARTIN96]中指出，导致一个软件设计的可维护性较低，也就是说会随着性能要求的变化而“腐烂”的真正原因有四个：过于僵硬（Rigidity）、过于脆弱（Fragility）、复用率低（Immobility）、黏度过高（Viscosity）。

过于僵硬

很难在一个软件系统里加入一个新的性能，哪怕是很小的都很难。这是因为加入一个新性能，不仅仅意味着建造一个独立的新模块，而且因为这个新性能会波及很多其他模块，最后变成跨越几个模块的改动。使得一个起初只需要几天的工作，最后演变成持续两个月的连续作战。

由于这种设计上的缺陷，使得项目经理不敢轻易向系统加入新功能。这就造成一个软件系统一旦做好，就不能增加新功能的僵硬化情况。

过于脆弱

与软件过于僵硬同时存在的，是软件系统在修改已有代码时过于脆弱。对一个地方的修改，往往会导致看上去没有什么关系的另一个地方发生故障。尽管在修改之前，设计师们会竭尽所能预测可能的故障地点，但是在修改完成之前，系统的原始设计师们甚至都无法确切预测到可能会波及到的地方。

这种一碰就碎的情况，造成软件系统过于脆弱。

复用率低

所谓复用，就是指一个软件的组成部分，可以在同一个项目的不同地方甚至另一个项目中重复使用。

每当程序员发现一段代码、函数、模块所做的事情是可以在新的模块、或者新系统中使用的时候，他们总是发现，这些已有的代码依赖于一大堆其他的东西，以至于很难将它们分开。最后，他们发现最好的办法就是不去“碰”这些已有的东西，而是重新写自己的代码。他们可能会使用源代码剪贴的办法，以最原始的复用方式，节省一些时间。

这样的系统就有复用率低的问题。

黏度过高

有的时候，一个改动可以以保存原始设计意图和原始设计框架的方式进行，也可以以破坏原始意图和框架的方式进行。第一种办法无疑会对系统的未来有利，第二种办法是权宜之计，可以解决短期的问题，但是会牺牲中长期的利益。

如果第二种办法比第一种办法要容易很多的话，程序员就有可能牺牲中长期的利益，采取权宜之计：在模块中搭建一个短路桥，或者在一个通用的逻辑中制造一个特例，以便解决眼前的需要。

一个系统设计，如果总是使得第二种办法比第一种办法容易，就叫做黏度过高。一个黏度过高的系统会诱使维护它的程序员采取错误的维护方案，并惩罚采取正确维护方案的程序员。

设计的目标

根据 Peter Coad 的[COAD99]一文，一个好的系统设计应该有如下的性质：可扩展性（Extensibility）、灵活性（Flexibility）、可插入性（Pluggability）。这三条性质就是一个系统设计应当达到的目标，下面就逐条讨论这三个目标。

可扩展性

新的性能可以很容易地加入到系统中去，就是可扩展性。这就是 Martin 所说的系统“过于僵硬”的属性的反面。

比如一个新的制动防滑系统应该可以在不影响汽车其他部分的情况下加入到系统中。如果加入了这个防滑系统之后，汽车的方向盘开始因此出问题，那么这个系统就不是扩展性很好的系统。

灵活性

可以允许代码修改平稳地发生，而不会波及到很多其他的模块，这就是灵活性。灵活性其实就是 Robert C. Martin 所说的“过于脆弱”的属性的反面。

比如一辆汽车的空调发生了故障，技师修理了空调。如果空调修好之后，发现系统的发动机不能启动了，这就不是一个很灵活的系统。

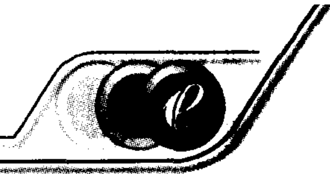
可插入性

可以很容易地将一个类抽出去，同时将另一个有同样接口的类加入进来，这就是可插入性。其实，这就是 Robert C. Martin 所说的系统“黏度过高”的反面。

比如应该可以很容易地将一辆汽车的防撞气囊取出来，换上一个新的。如果将气囊拿出来后，汽车的传动杆不工作了，那么这个系统就不是一个可插入性很好的系统。

读者可能会觉得，这些汽车的比喻都很荒唐：加入防滑系统怎么会影响到汽车的方向盘呢？修理了空调，怎么会使得发动机不能启动呢？换一个气囊，怎么会使传动杆停止工作呢？可是如果读者想一想自己所经历的软件系统的设计，就不会觉得荒唐了，因为软件系统中的问题比这些荒唐的比喻还要荒唐。

那么，怎样才能做出一个符合这三项要求的设计呢？关键是恰当地提高软件的可维护性和可复用性。



3.2　系统的可复用性

“复用”（Reuse）有时译作“重用”，是重复使用的意思。自从 20 世纪 60 年代以来，软件的复用便是重要研究的课题，但是一般软件的复用率尚不高，在国内的软件界尤其需要改善。

复用的重要性

软件的复用的好处有：第一，较高的生产效率；第二，较高的软件质量；第三，恰当使用复用可以改善系统的可维护性。

一个可以重复使用的软件成分可以为将来的使用节省费用。一个构件被复用的频率越高，构件的初始开发投资就相对越少。

一个可复用的软件成分总是比不能复用的软件成分有更多的质量保证。如果一个复用率高的软件构件有程序缺陷的话，这种缺陷可以更快地、更彻底地被排除。这样的软件成分必定是有利于系统的可维护性的。

更为重要的是，复用与系统的可维护性有直接的关系，这主要是由于在面向对象的系统设计中复用的概念与传统的概念有很大的不同。这一点会在下面具体讲述。

传统的复用

复用不仅仅是代码的复用，虽然代码复用确实是复用的一个初等形式。

代码的剪贴复用

利用现代的代码编辑器、集成开发环境（IDE）等，可以很容易地做到减少抄写代码的人力成本。这虽然比完全没有复用好一些，但是靠剪贴源代码达到复用，仅仅是复用的最初步的做法。

源代码的剪贴往往作为程序员自发的代码复用形式出现。在具体实施时，仍然要冒着产生错误的风险。管理人员不可能跟踪大块代码的变异和使用。

由于同样或类似的源代码同时被复制到多个软件成分中，当这块软代码发生程序错误需要修改时，程序员需要独立地修改每一块拷贝。由于同样的原因，多个软件成分需要独立地检测。复用所能节省的初期投资十分有限。

算法的复用

各种算法比如排序算法得到了大量的研究。现在几乎不会有人在应用程序编程时试图建立自己的排序算法，通常的做法是在得到了很好的研究的各种算法中选择一个。这就是算法的复用。

数据结构的复用

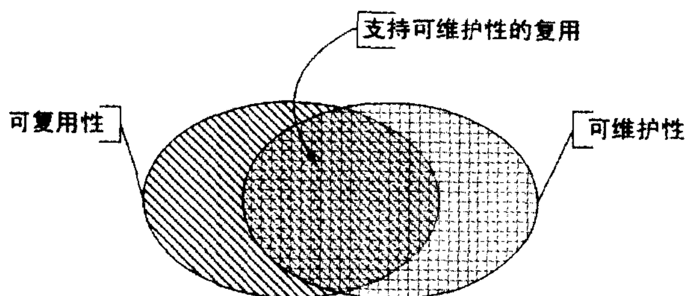
与算法的复用相对的，是数据结构的复用。如队、站、队列、列表等数据结构得到了

十分透彻的研究，所有的计算机教学都要详细描述这些数据结构。这就是数据结构的复用。

这些都是传统的复用种类，它们都各有优点，也各有缺陷。

可维护性与复用的关系

传统的复用方案的一个致命缺陷就是复用常常是以破坏可维护性为代价的[KIRK02]。比如两个模块 A 和 B 同时使用另一个模块 C 中的功能。那么当 A 需要 C 增加一个新的行为的时候，B 有可能不需要、甚至不允许 C 增加这个新行为。如果坚持使用复用，就不得不以系统的可维护性为代价；而如果从保持系统的可维护性出发，就只好放弃复用。可维护性与可复用性是有共同性的两个独立特性，它们就像是两只同时在奔跑的兔子，如下图所示。



因此，一个重要的概念就是支持可维护性的复用，也就是在保持甚至提高系统的可维护性的同时，实现系统的复用。那么怎样才能设计一个系统，以达到提高可维护性的复用目的呢？换言之，怎样才能抓住这两只同时在奔跑的兔子呢？

面向对象设计的复用

在一个像 Java 这样的面向对象的语言中，数据的抽象化、继承、封装和多态性是几项最重要的语言特性，这些特性使得一个系统可以在更高的层次上提供可复用性。数据的抽象化和继承关系使得概念和定义可以复用；多态性使得实现和应用可以复用[PALS90]；而抽象化和封装可以保持和促进系统的可维护性。这样一来，复用的焦点不再集中在函数和算法等具体实现细节上，而是集中在最重要的含有宏观商业逻辑的抽象层次上。换言之，复用的焦点发生了“倒转”。

发生复用焦点的倒转并不是因为实现细节的复用不再重要，而是因为这些细节上的复用往往已经做得很好，而且抽象层次是比这些细节更值得强调的复用焦点，因为它们是在提高复用性的同时保持和提高可维护性的关键。

既然抽象层次是一个应用系统做战略性判断和决定的地方，那么抽象层次就应当是较为稳定的，应当是复用的重点。如果抽象层次的模块相对独立于具体层次的模块的话，那么具体层次内部的变化就不会影响到抽象层次的结构，所以抽象层次的模块的复用就会较为容易。

在面向对象的设计里面，可维护性复用是以设计原则和设计模式为基础的，请参见下

面的分析。

对可维护性的支持

仍然以 Peter Coad 所给出的三个设计目标为出发点进行阐述。

首先，恰当地提高系统的可复用性，可以提高系统的可扩展性。允许一个具有同样接口的新的类代替旧的类，是对抽象接口的复用。客户端依赖于一个抽象的接口，而不是一个具体实现类，使得这个具体类可以被另一个具体类所取代，而不影响到客户端。

系统的可扩展性是由“开–闭”原则、里氏代换原则、依赖倒转原则和组合/聚合复用原则所保证的。

其次，恰当地提高系统的可复用性，可以提高系统的灵活性。在一个设计得当的系统中，每一个模块都相对于其他模块独立存在，并只保持与其他模块的尽可能少的通信。这样一来，在其中某一个模块发生代码修改的时候，这个修改的压力不会传递到其他的模块。

系统的灵活性是由“开–闭”原则、迪米特法则、接口隔离原则所保证的。

最后，恰当地提高系统的可复用性，可以提高系统的可插入性。在一个符合“开–闭”原则的系统中，抽象层封装了与商业逻辑有关的重要行为，这些行为的具体实现由实现层给出。当一个实现类不再满足需要，需要以另一个实现类取代的时候，系统的设计可以保证旧的类可以被“拔出（Unplug）”，新的类可以被“插入（Plug）”。

系统的可插入性是由“开–闭”原则、里氏代换原则、组合/聚合复用原则以及依赖倒转原则保证的。

这就是说，本书将要介绍的这些设计原则是在提高一个系统的可维护性的同时，提高这个系统的可复用性的指导原则。依照这些设计原则进行系统设计，就可以抓到这两只同时在奔跑的兔子。

设计原则有哪些

本书在后面将要介绍的设计原则如下：

- “开–闭”原则（Open-Closed Principle，或者 OCP）
- 里氏代换原则（Liskov Substitution Principle，或者 LSP）
- 依赖倒转原则（Dependency Inversion Principle，或者 DIP）
- 接口隔离原则（Interface Segregation Principle，或者 ISP）
- 组合/聚合复用原则（Composition/Aggregation Principle，或者 CARP）
- 迪米特法则（Law of Demeter，或者 LoD）

这些设计原则首先都是复用的原则，遵循这些设计原则可以有效地提高系统的复用性，同时提高系统的可维护性。

学习设计模式对复用性与可维护性的帮助

凡是有理论的地方，就有如何恰当地将理论应用到实践中去的问题，设计模式是对学习 OO 设计原则的具体指导。

本书将要介绍的设计模式可以划分成创建模式、结构模式和行为模式三大类别，专门讲解的设计模式有 26 个，在讲解过程中所涉及到的设计模式就更多。

设计模式本身并不能保证一个系统的可复用性和可维护性，但是通过学习这些设计模式的思想可以有效提高设计师的设计风格、设计水平，并促进与同行之间的沟通，从而帮助设计师提高系统设计的复用性和可维护性。

3.3 老子论“不武”

《老子》云：“善为士者不武”。如果“士”就是软件系统设计师，“武”就是对软件系统的大规模修改，那么老子说的就可以解释为好的设计师不会在他设计的系统投入使用后再进行大规模的修改。老子的所谓“不武”，便是软件设计中的“复用”。

《老子》还说：“天下有道，却走马以粪，天下无道，戎马生于效。”也就是说，当天下有道（太平）时，好的跑马却在田间耕作；当天下无道（不太平）时，战马在战场上生出小驹。

这相当于说，当一个软件系统是一个复用有道、易于维护的系统时，将新的性能加入到系统中去，或者对一个已有的性能进行修改是不困难的事情，因此，代码高手就无法发挥作用；而当一个软件系统是一个设计低劣、可维护性很差的系统时，代码高手就必须连续作战，才能将新的性能加入到系统中去，或者对一个已有的性能进行修改。

中国古代的思想大师对很多问题的哲学论述，并没有随着时间的转移而失去其影响力。这些论述的精辟和洞察力，都使得它们可以应用到软件设计理论中去，从而发挥意想不到的威力。

参考文献

[MARTIN96] Robert C. Martin. Engineering Notebook——C++ Report. Nov-Dec，1996

[MARTIN00]Robert C. Martin. Design Principles and Design Patterns. www.objectmentor.com，2000

[KIRK02]Kirk Knoernschild. Java Design – Objects，UML，and Process. Addison-Wesley，2002

[WR00] Walker Royce. Next-Generation Software Economics. The Rational Edge，December 2000

[COAD99] Peter Coad，Mark Mayfield and Jon Kern. Java Design – Building Better Apps and Applets. Prentice Hall，1999

[PALS90]J. Palsberg and M. Schwartzbach. Type Substitution in Object-Oriented Programming. OOPSLA/ECOOP'90 Conference Proceedings，October 1990

第 4 章 “开–闭”原则（OCP）

经典力学的基石是牛顿三大定律。而面向对象的可复用设计（Object Oriented Design, 或 OOD）的第一块基石，便是所谓的“开–闭”原则（Open-Closed Principle，常缩写为 OCP）。

4.1 什么是“开–闭”原则

“开–闭”原则讲的是：一个软件实体应当对扩展开放，对修改关闭。这一原则最早由 Bertrand Meyer [MEYER88]提出，英文原文是：

Software entities should be open for extension, but closed for modification.

这个原则说的是，在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。换言之，应当可以在不必修改源代码的情况下改变这个模块的行为。

这话听上去好像是矛盾的，但是实际上本书将在后面给出的几个面向对象的设计原则和设计模式中很具体地说明如何在设计上做到这一点。

所有的软件系统都有一个共同的性质，即对它们的需求都会随时间的推移而发生变化。在软件系统面临新的需求时，系统的设计必须是稳定的。满足“开–闭”原则的设计可以给一个软件系统两个无可比拟的优越性：

- 通过扩展已有的软件系统，可以提供新的行为，以满足对软件的新需求，使变化中的软件系统有一定的适应性和灵活性。
- 已有的软件模块，特别是最重要的抽象层模块不能再修改，这就使变化中的软件系统有一定的稳定性和延续性。

具有这两个优点的软件系统是一个在高层次上实现了复用的系统，也是一个易于维护的系统。

4.2 怎样做到“开–闭”原则

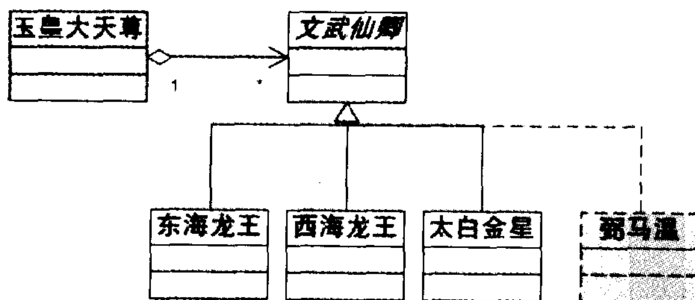
乍看起来，不能修改而可以扩展似乎是自相矛盾的。怎么可以同时又不修改、而又可扩展呢？在开始讨论之前，不妨考察一下《西游记》中，玉皇大帝在美猴王的挑战之下，是怎样维护天庭的秩序的。

玉帝招安美猴王

当年大闹天宫时的美猴王便是玉帝天庭的新挑战。美猴王说：“‘皇帝轮流做，明年到我家。’只教他搬出去，将天宫让与我！”对于这项挑战，太白金星给玉皇大帝提出的建议

是："臣启陛下……降一道招安圣旨，把他宣来上界……与他籍名在箓……一则不动众劳师，二则收仙有道也。"

换言之，不劳师动众、不破坏天规便是"闭"，收仙有道便是"开"。招安之法便是玉帝天庭的"开-闭"原则，通过给美猴王封一个"弼马温"的官职，便可使现有系统满足变化了的需求，而不必更改天庭的既有秩序，如下图所示。



招安之法的关键便是不允许更改现有的天庭秩序，但允许将妖猴纳入现有秩序中，从而扩展了这一秩序。用面向对象的语言来讲，不允许更改的是系统的抽象层，而允许扩展的是系统的实现层。

《太玄》论"固革"

西汉杨雄的《太玄》一书说："知固而不知革，物失其则；知革而不知固，物失其均。"

一个系统对修改关闭，就是《太玄》所说的"固"；而系统对扩展开放，就是《太玄》所说的"革"。一个系统不可扩展，就会"物失其则"，或者说系统失去使用的价值；而一个系统动辄需要修改，便会"物失其均"，也就是失去重心。

因此，"开-闭"原则非常接近《太玄》一书所说的"固革"原则。

抽象化是关键

解决问题的关键在于抽象化。在像 Java 语言这样的面向对象的编程语言里面，可以给系统定义出一个一劳永逸、不再更改的抽象设计，此设计允许有无穷无尽的行为在实现层被实现。在 Java 语言里，可以给出一个或多个抽象 Java 类或 Java 接口，规定出所有的具体类必须提供的方法的特征（Signature）作为系统设计的抽象层。这个抽象层预见了所有的可能扩展，因此，在任何扩展情况下都不会改变。这就使得系统的抽象层不需修改，从而满足了"开-闭"原则的第二条：对修改关闭。

同时，由于从抽象层导出一个或多个新的具体类可以改变系统的行为，因此系统的设计对扩展是开放的，这就满足了"开-闭"原则的第一条。

关于在 Java 语言中实现抽象化的方法，请阅读后面的"专题：Java 接口"一章和"专题：抽象类"一章。



对可变性的封装原则

“开–闭”原则如果从另外一个角度讲述，就是所谓的“对可变性的封装原则”（Principle of Encapsulation of Variation，常常略写做 EVP）。“对可变性的封装原则”讲的是找到一个系统的可变因素，将之封装起来。

在[GOF95]中说：考虑你的设计中什么可能会发生变化。与通常将焦点放到什么会导致设计改变的思考方式正好相反，这一思路考虑的不是什么会导致设计改变，而是考虑你允许什么发生变化而不让这一变化导致重新设计。

[SHALL01]将这一思想用一句话总结为：“找到一个系统的可变因素，将它封装起来”，并将它命名为“对可变性的封装原则”。

“对可变性的封装原则”意味着两点：

（1）一种可变性不应当散落在代码的很多角落里，而应当被封装到一个对象里面。同一种可变性的不同表象意味着同一个继承等级结构中的具体子类，因此，读者可以期待在设计模式中看到继承关系。

继承应当被看做是封装变化的方法，而不应当被认为是从一般的对象生成特殊的对象的方法[SHALL01]。

（2）一种可变性不应当与另一种可变性混合在一起。如果读者留心本书所研究的这些设计模式的类图的话，就会发现所有的类图的继承结构一般都不会超过两层，不然就意味着将两种不同的可变性混合在了一起。

显然，“对可变性的封装原则”从工程的角度讲解了如何实现“开–闭”原则。如果能够将“对可变性的封装原则”作为总的设计原则的话，那么按照这个原则进行的系统设计，应当遵守“开–闭”原则。

尽管在很多情况下，无法百分之百地做到“开–闭”原则，但是如果向这个方向上的努力能够得到部分的成功，也可以显著地改善一个系统的结构。

4.3 与其他设计原则的关系

做到“开–闭”原则不是一件容易的工作，但是也是有很多规律可循的。这些规律也同样以设计原则的身份出现，但是它们都是“开–闭”原则的手段和工具，是附属于“开–闭”原则的。

里氏代换原则

里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。

里氏代换原则是对“开–闭”原则的补充。正如前面所谈到的，实现“开–闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体体现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

一般而言，违反里氏代换原则的，也违背“开–闭”原则，反过来并不一定成立。

依赖倒转原则

依赖倒转原则讲的是，要依赖于抽象，不要依赖于实现。

看上去依赖倒转原则与“开–闭”原则有很大的相似之处，实际上，它们之间的关系是目标和手段之间的关系。“开–闭”原则是目标，而达到这一目标的手段是依赖倒转原则。

换言之，要想实现“开–闭”原则，就应当坚持依赖倒转原则。违反依赖倒转原则，就不可能达到“开–闭”原则的要求。

合成/聚合复用原则

合成/聚合复用原则讲的是，要尽量使用合成/聚合，而不是继承关系达到复用的目的。

显然，合成/聚合复用原则是与里氏代换原则相辅相成的，两者又都是对实现“开–闭”原则的具体步骤的规范。前者要求设计师首先考虑合成/聚合关系，后者要求在使用继承关系时，必须确定这个关系是符合一定条件的。

遵守合成/聚合复用原则是实现“开–闭”原则的必要条件；违反这一原则就无法使系统实现“开–闭”原则这一目标。

迪米特法则

迪米特法则讲的是，一个软件实体应当与尽可能少的其他实体发生相互作用。

当一个系统面临功能扩展的时候，其中会有一些模块，它们需要修改的压力比其他一些模块要大。最后的结果可能是这些模块需要修改或者不需要修改。但是不论是哪一种情况，如果这些模块是相对孤立的，那么它们就不会将修改的压力传递给其他的模块。

这就是说，一个遵守迪米特原则设计出来的系统在功能需要扩展时，会相对更容易地做到对修改的关闭。也就是说，迪米特法则是一条通向“开–闭”原则的道路。

接口隔离原则

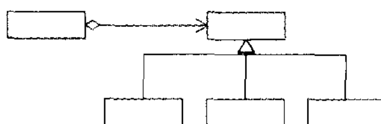
接口隔离原则讲的是，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。

显然，接口隔离原则与广义的迪米特法则都是对一个软件实体与其他的软件实体的通信的限制。广义的迪米特法则要求尽可能限制通信的宽度和深度。接口隔离原则所限制的是通信的宽度，也就是说，通信应当尽可能地窄。

显然，遵循接口隔离原则与迪米特法则，会使一个软件系统在功能扩展的过程当中，不会将修改的压力传递到其他的对象。

4.4 策略模式对“开–闭”原则的支持

策略模式讲的是，如果有一组算法，那么就将每一个算法封装起来，使得它们可以互换。显然，策略模式就是从对可变性的封装原则出发，达到“开–闭”原则的一个范例，如下图所示。



在本书后面的“策略（Strategy）模式”一章讨论了图书销售的折扣计算问题。显然根据描述，折扣是根据以下的几个算法中的一个进行的：

算法一：对有些图书没有折扣。折扣算法对象返还 0 作为折扣值。

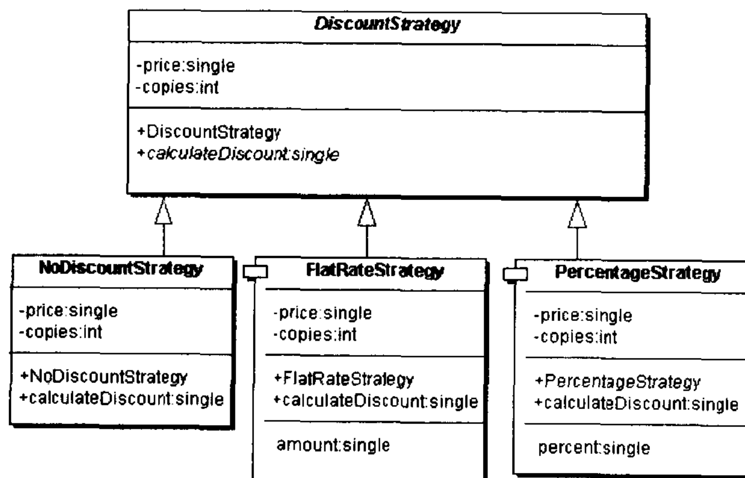
算法二：对有些图书提供一个固定量值为 1 元的折扣。

算法三：对有些图书提供一个百分比的折扣，比如一本书价格为 20 元，折扣百分比为 7%，那么折扣值就是 20*7%=1.4（元）。

在采用策略模式之前，设计师必须从“开–闭”原则出发，考察这个图书销售系统是否有可能在将来引入新的折扣算法。如果确有可能，那么就应当将所有的折扣算法封装起来，因为它们是可变化的因素。系统必须能够在新的算法出现时，很方便地将新的算法插入到已有的系统中，而不必修改已有的系统。

使用策略模式描述的话，这些不同的算法都是不同的具体策略角色。从算法一到算法三分别可以用 NoDiscountStrategy 对象，FlatRateStrategy 对象和 PercentageStrategy 对象描述。

为了使这些算法成为“即插即用（Pluggable）”的对象，必须使这些算法能够相互替换。而做到这一点的关键，就是给这些对象定义出相同的接口，也就需要有一个抽象策略角色作为这些对象所组成的等级结构的超类型。在现在所讨论的图书销售的折扣计算系统中，这个抽象策略角色由一个 Java 抽象类 DiscountStrategy 扮演，如下图所示。



关于这个例子的详细解释，请读者阅读本书的“策略（Strategy）模式”一章。

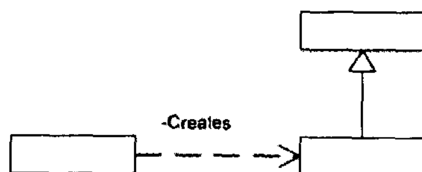
4.5 在其他设计模式中的体现

[GOF95]指出，“对可变性的封装原则”实际上是设计模式的主题。换言之，所有的设计模式都是对不同的可变性的封装，从而使系统在不同的角度上达到“开–闭”原则的要求。

下面就给出几个浅显的例子，结合相关的模式讲解这个 OO 设计原则。

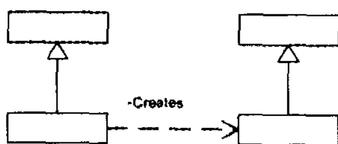
简单工厂模式

“开–闭”原则要求系统允许新的产品加入系统中，而无需对现有代码进行修改。在简单工厂模式中，这对于产品的消费角色是成立的，而对于工厂角色是不成立的，如下图所示。每次增加一个新的产品，都需要修改工厂角色。但是产品的消费者则可以避免进行修改。



工厂方法模式

在工厂方法模式中，具体工厂类都有共同的接口，它们“生产”出很多的处于一个等级结构中的产品对象。使用这个设计的系统可以允许向系统加入新的产品类型，而不必修改已有的代码，只需要再加入一个相应的新的具体工厂类就可以了。工厂方法模式的简略类图如下图所示。

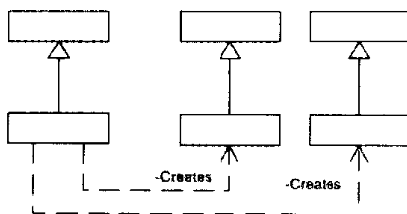


换言之，对于增加新的产品类而言，这个系统完全支持“开–闭”原则。

抽象工厂模式

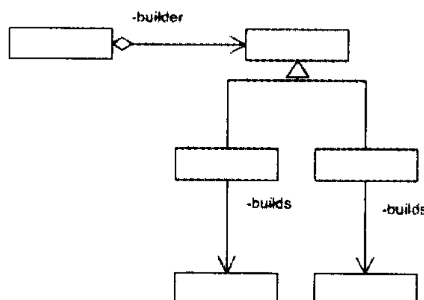
抽象工厂模式封装了产品对象家族的可变化性，从而一方面可以使系统动态地决定将哪一个产品族的产品实例化，另一方面可以在新的产品对象引进到已有的系统中时不必修

改已有的系统。换言之，在产品对象家族发生变化时，这一设计可以维持系统的“开-闭”性。抽象工厂模式的简略类图如下图所示。



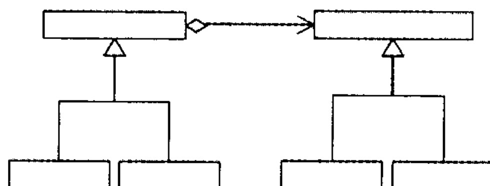
建造模式

建造模式封装了建造一个有内部结构的产品对象的过程，因此，这样的系统是向产品内部表象的改变开放的。建造模式的简略类图如下图所示。



桥梁模式

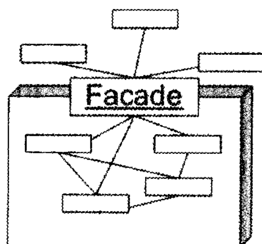
桥梁模式是“对可变性的封装原则”的极好例子。在桥梁模式中，具体实现化类代表不同的实现逻辑，但是所有的具体实现化类又有共同的接口。新的实现逻辑可以通过创建新的具体实现化类加入到系统里面。桥梁模式的简略类图如下图所示。



门面模式

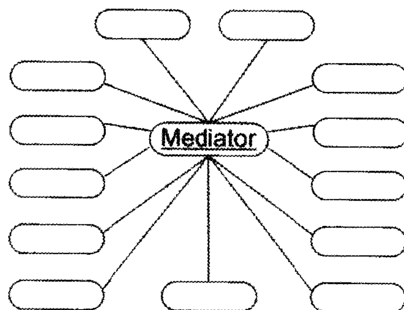
假设一个系统开始的时候与某一个子系统耦合在一起，后来又不得不换成另外一个子系统，那么门面模式便可以发挥门面模式和适配器模式两种作用，将新的子系统仍然与本

系统耦合在一起。这样一来，使用门面模式便可以改变子系统内部功能而不会影响到客户端。门面模式的简略类图如下图所示。



调停者模式

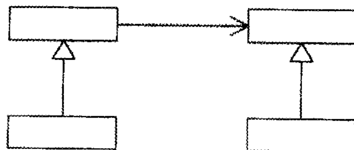
调停者模式使用一个调停者对象协调各个同事对象的相互作用，这些同事对象不再发生直接的相互作用。调停者模式的简略类图如下图所示。



这样，一旦有新的同事对象添加到系统中来的时候，这些已有的同事对象都不会受到任何影响，但是调停者对象本身却需要修改。换言之，调停者模式以一种不完美的方式支持“开–闭”原则。

访问者模式

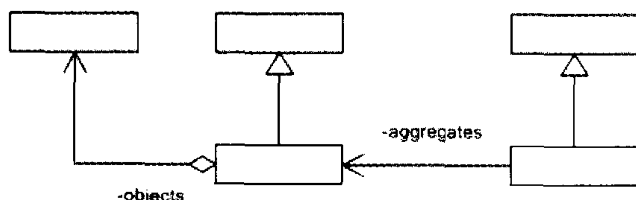
访问者模式使得在节点中加入新的方法变得很容易，仅仅需要在一个新的访问者类中加入此方法就可以了，但是访问者模式不能很好地处理增加新的节点的情况。换言之，访问者模式提供了倾斜的可扩展性设计：方法集合的可扩展性和类集合的不可扩展性。访问者模式的简略类图如下图所示。



也就是说，访问者模式的使用可以使一个节点系统对方法集合的扩展开放。

迭代子模式

迭代子模式将访问聚集元素的逻辑封装起来，并且使它独立于聚集对象的封装。这就提供了聚集存储逻辑与迭代逻辑独立演变的空间，使系统可以在无需修改消费迭代子的客户端的情况下，对聚集对象的内部结构进行功能扩展。迭代子模式的简略类图如下图所示。



当读者学习设计模式的时候，要学会问一个问题：这个设计模式可以对什么样的变换开放，以及它做到这一点所付出的代价是什么。通过这样的思考，可以更加透彻地了解这种模式对“开–闭”原则的支持程度，以及这种设计模式本身。

4.6 一个重构做法的讨论

“将条件转移语句改写成为多态性”是一条广为流传的代码重构做法。它的意思是说，将一个进行多次条件转移的商业逻辑封装到不同的具体子类中去，从而使用多态性代替条件转移语句。

本书下面就从“开–闭”设计原则的角度出发分析这一代码重构做法的适用性。

寻找可变性的线索

首先，这一代码重构的做法是实现“开–闭”原则的一条重要线索，因为条件转移语句特别是大段大段的代码转移语句块往往意味着某种可变性。将这种可变性用多态性代替，就意味着将这种可变性封装起来，从而带来系统在这种变化发生时的“开–闭”特性。

但是，这一做法本身并不能保证实现“开–闭”原则，设计师应当以“开–闭”原则为指导原则。这一代码重构的做法并不能成为设计的原则。事实上，这一做法有明显的缺点：

- 任何语言都提供条件转移功能，条件转移本身并不是错误的，更不是什么罪恶。如果需要，设计师完全可以选择使用条件转移。
- 使用多态性代替条件转移意味着大量的类被创建出来。比如，一个类如果有三个方法，每个方法都有一个三段的条件转移语句。如果将它们都用多态性代替的话，就会造成九个不同的类。很难想像设计师怎样能明白这九种组合成员之间的关系。

何时使用这种重构做法

那么设计师应当怎样判断何时使用多态性取代条件转移语句呢？回答是应当从“开–闭”原则出发来做判断。如果一个条件转移语句确实封装了某种商务逻辑的可变性，那么将此种可变性封装起来就符合“开–闭”原则设计思想了。

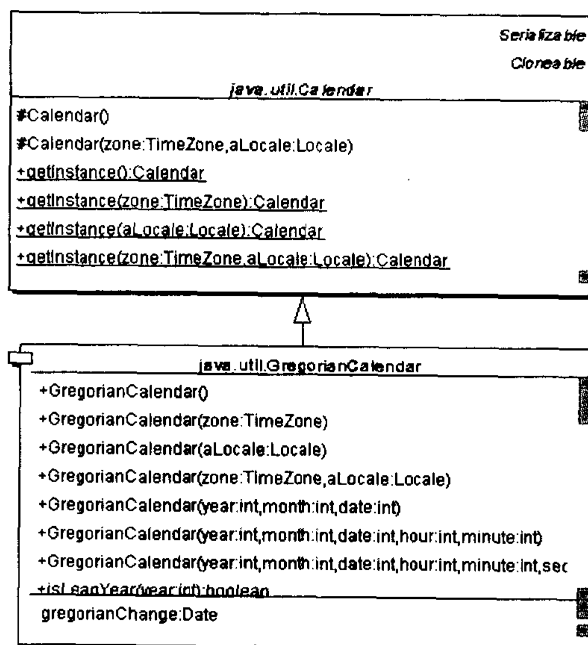
但是，如果一个条件转移语句没有涉及重要的商务逻辑，或者不会随着时间的变化而变化，也不意味着任何的可扩展性，那么它就没有涉及任何有意义的可变性。这时候将这个条件转移语句改写成为多态性就是一种没有意义的浪费。本书将这种对多态性的滥用叫做“多态性污染”。

问答题

请问 java.util.Calendar 符合“开–闭”原则吗？

问答题答案

java.util.Calendar 是一个抽象类，它应当给出一个不仅仅适用于格里高利历法的方法和公开的常量，而且应当给出适用于所有的历法，包括中国的阴历的方法和常量。比如，大家使用的公元历法属于格里高利历法，日期和时间的运算都应当使用 GregorianCalendar 类，如下图所示。



但是，java.util.Calendar 却定义出只适用于格里高利历法的方法和公开的常量，如代码清单 1 所示。

代码清单 1：Calendar 类的部分源代码

```
public final static int SUNDAY = 1;
    public final static int MONDAY = 2;
    public final static int TUESDAY = 3;
    public final static int WEDNESDAY = 4;
    public final static int THURSDAY = 5;
    public final static int FRIDAY = 6;
    public final static int SATURDAY = 7;
    public final static int JANUARY = 0;
    public final static int FEBRUARY = 1;
    public final static int MARCH = 2;
    public final static int APRIL = 3;
    public final static int MAY = 4;
    public final static int JUNE = 5;
    public final static int JULY = 6;
    public final static int AUGUST = 7;
    public final static int SEPTEMBER = 8;
    public final static int OCTOBER = 9;
    public final static int NOVEMBER = 10;
    public final static int DECEMBER = 11;
    /**
     * Sets what the first day of the week is; e.g., Sunday in US,
     * Monday in France.
     */
    public void setFirstDayOfWeek(int value)
    {
        firstDayOfWeek = value;
    }
    /**
     * Gets what the first day of the week is; e.g., Sunday in US,
     * Monday in France.
     */
    public int getFirstDayOfWeek()
    {
        return firstDayOfWeek;
    }
    public void setMinimalDaysInFirstWeek(int value)
    {
        minimalDaysInFirstWeek = value;
    }
    public int getMinimalDaysInFirstWeek()
    {
        return minimalDaysInFirstWeek;
    }
```

这些英文星期名称与月份名称的常量对中国阴历而言没有任何意义。阴历以十天为一

周，因此，java.util.Calendar 抽象类关于星期的运算并不适用于中国的阴历，而仍然仅适用于格里高利历法。

换言之，Calendar 类没有办法容纳中国阴历，所以并不支持“开–闭”原则。

参考文献

[MEYER88] Bertrand Meyer. Object Oriented Software Construction. Prentice Hall，1988. P23

[MARTIN96] Robert C. Martin. The Open-Closed Principle，Engineering Notebook——C++ Report. Jan，1996

[MARTIN96] Robert C. Martin. Engineering Notebook——C++ Report. Nov-Dec，1996

[WORDEN00a] Brent Worden. The Open-Closed Principle. http://www.brent.worden.org

[SHALL01] Alan Shalloway，James R. Trott. Design Patterns Explained – A New Perspective on Object-Oriented Design. Addison–Wesley，2001

[FOWLER01] Martin Fowler . Refactoring——Improving the Design of Existing Code. Addison–Wesley，2001.

[KIRK02] Kirk Knoernschild. Java Design – Objects，UML，and Process. Addison-Wesley，2002

[YANG] 杨雄（西汉）著.《太玄经》

第 5 章　专题：Java 语言的接口

在家中，可以很容易地将微波炉从一个电源插座上拔下，然后将手提电脑插上去。这就是说，对于电源来说，这些电器都是可插入构件（Pluggable Component）。它们之所以是可插入的（Pluggable），是因为它们都具有与电源插座相匹配的插头。

如果可以动态地将一个构件移走，并以另一个构件取而代之，那么这种构件就是可插入构件。

读者可以看出，所谓的接口，实际上就相当于电源插座；而可插入构件就相当于这种插座匹配的电器。正如电器的可插入性是由统一的电源插座及每个电器都有一个与之匹配的电源插头造成的一样，可插入构件的关键在于存在一个公用的接口，以及每个构件都实现了这个接口。

因此，接口是实现构件的可插入性（Pluggability）的关键。

5.1　什么是接口

一个 Java 接口（Interface）是一些方法特征的集合，这些方法特征当然来自于具体方法，但是它们一般都是来自于一些在系统中不断出现的方法。一个接口只有方法的特征，而没有方法的实现，因此这些方法在不同的地方被实现时，可以具有完全不同的行为。在 Java 语言中，Java 接口还可以定义 public 的常量。

在人们讲到“接口”的时候，这个词往往有两种不同的含义：第一种是指 Java 接口，这是一种 Java 语言中存在的结构，有特定的语法和结构；另一种仅仅是指一个类所具有的方法的特征集合，是一种逻辑上的抽象。前者叫做“Java 接口”，后者就叫做“接口”。比如，java.lang.Runnable 就是一个 Java 接口，它的源代码如代码清单 1 所示。

代码清单 1：Runnable 接口的源代码

```
public interface Runnable
{
  public abstract void run();
}
```

在任何可能出现混淆的地方，本书都会特别予以指出。

在 Java 语言规范（Java Language Specification）[JLS00]中，一个方法的特征仅包括方法的名字、参量的数目和种类，而不包括方法的返还类型、参量的名字以及所抛出的异常。在 Java 编译器检查方法的重载（Overload）时，会根据这些条件判断两个方法是否是重载方法。但是在 Java 编译器检查方法的置换（Override）时，则会进一步检查两个方法（分处超类型和子类型）的返还类型和抛出的异常是否相同。

在使用接口时，需要指明接口本身，以及实现这个接口的类。一个类实现一个接口，这种关系叫做接口继承（Interface Inheritance）；而一个类是另一个类的子类，这种关系叫做实现继承（Implementation Inheritance）。接口继承的规则与实现继承的规则不同，一个类最多只能够有一个超类，但是可以同时实现几个接口。

Java 接口本身没有任何实现，因为 Java 接口不涉及表象，而只描述 public 行为，所以 Java 接口比 Java 抽象类更为抽象化。

一个 Java 接口的方法只能是抽象的和公开的，Java 接口不能有构造子。Java 接口可以有 public、静态的和 final 的属性。

实现一个共同的 Java 接口的两个类常常完全不同，但是有一组相同的方法和常数。一些看上去毫不相干的类，可以因为提供相类似的服务，从而具有相同的接口。比如说，两个实现 Runnable 接口的类除了都有一个 public void run() 方法之外，可能没有其他任何的共同点。

接口和类的最重要区别是，接口仅仅描述方法的特征（Signature），而不给出方法的实现；而类不仅给出方法的特征，而且给出方法的实现。因此，接口把方法的特征和方法的实现分割开来。这种分割，体现在接口常常代表一个角色（role），它包装与该角色相关的操作和属性，而实现这个接口的类便是扮演这个角色的演员。一个角色可以由不同的演员来演，而不同的演员之间除了扮演一个共同的角色之外，并不要求有任何其他的共同之处。

5.2 为什么使用接口

首先，如果没有接口会怎么样呢？一个类总归可以从另一个类继承，难道这还不够吗？

没有接口会怎样

一个对象需要知道其他的一些对象，并且与其他的对象发生相互作用，这是因为这些对象需要借助于其他对象的行为以便完成一项工作。这些关于其他对象的知识，以及对其他对象行为的调用，都是使用硬代码（Hardcode）写在类里面的，可插入性几乎是零。

如果要加入一个新的类，仅仅意味着加入新的硬代码，而不能给出动态的可插入性。

那么基于类的继承是否可以提供一点帮助呢？比如现在有一个具体类，提供某种使用硬代码写在类中的行为。现在，要提供一些类似的行为，并且实现动态的可插入，也就是说，要能够动态地决定使用哪一种实现。一个可能的做法就是为这个类提供一个抽象超类，声明出子类要提供的行为，然后让这个具体类继承自这个抽象超类。同时，为这个抽象超类提供另一个具体子类，这个子类以不同的方法实现了超类所声明的行为。客户端可以动态地决定使用哪一个具体子类。这是否可以提供可插入性呢？

答案是，这确实可以在简单的情况下提供可插入性。但是由于 Java 语言是一个单继承的语言，换言之，一个类只能有一个超类，因此，在很多情况下，这个具体类可能已经有了一个超类，这时候，要给它加上一个新的超类是不可能的。如果硬要做的话，就只好



把这个新的超类加到已有的超类上面，形成超超类的情况；如果这个超超类的位置也已经被占用了，就只好继续向上移动，直到移动到类等级结构的最顶端。这样一来，对一个具体类的可插入性设计，就变成了对整个等级结构中所有的类的修改。

这还是假设这些超类都是在设计师控制之下的。如果这些超类是一些软件商提供的，设计师无法修改，怎么办呢？比如有一个具体类，它有一个超类是 Frame，那么新加入的超类就只好一直向上移动到 java.lang.Object 类上面，这怎么可能呢？

因此，没有接口，可插入性就没有保证。

接口是对可插入性的保证

接口使可插入性变得可能。

在一个类等级结构中的任何一类都可以实现一个接口，这个接口会影响到此类的所有子类，但是不会影响到此类的任何超类。此类将不得不实现这个接口所规定的方法，而其子类则可以从此类自动继承到这些方法，当然也可以选择置换掉所有的这些方法，或者其中的某一些方法。

这时候，这些子类就具有了可插入性。

关联的可插入性

正如前面所说的，一个对象需要完成一项任务，所以需要知道其他的对象，并且调用其他对象的方法。这个对象对其他对象的知识叫做关联（Association）。

如果一个关联不是针对一个具体类的，而是针对一个接口的，那么任何实现这个接口的类就都可以满足要求。换言之，当前对象并不在意所关联的是哪一个具体类，而仅仅关心这个类是否实现了某一个接口。

这样一来，就可以动态地将这个关联从一个具体类转换到另一个具体类，而这样做的惟一条件是它们都实现某个接口。

调用的可插入性

同样，一个对象不可避免地需要调用其他对象的方法。这种调用不一定非得是某一个具体类，而可以是一个接口。这样一来，任何实现了这个接口的具体类都可以被当前对象调用；而当前对象到底调用的是哪一个具体类的实例则完全可以动态地决定。

因此，接口提供了关联以及方法调用上的可插入性，软件系统的规模越大，生命周期越长，接口的重要性就越大。接口使得软件系统在灵活性和可扩展性、可插入性方面得到保证。

类型

Java 接口（以及 Java 抽象类）用来声明一个新的类型。

Java 设计师应当主要使用 Java 接口和抽象 Java 类将软件单位与内部和外部耦合起来。换言之，应当使用 Java 接口和抽象 Java 类而不是具体类进行变量的类型声明、参量的类型声明、方法的返还类型声明，以及数据类型的转换等。当然，一个更好的做法是仅仅使

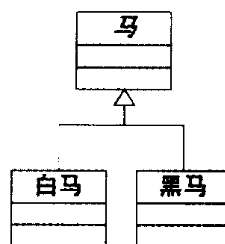
用 Java 接口，而不要使用抽象 Java 类来做到上面这些。

在理想的情况下，一个具体 Java 类应当只实现 Java 接口和抽象 Java 类中声明过的方法，而不应当给出多余的方法。

类型等级结构

Java 接口（以及 Java 抽象类）一般用来作为一个类型的等级结构的起点。

Java 的类型是以类型等级结构（Type Hierarchy）的方式组织起来的。在一个类型等级结构里面，一个类型可以有一系列的超类型（Supertype），这时这个类型叫做其超类型的子类型。比如白马和黑马均是马的子类型，而马作为一个超类型是这个等级结构的第一层，如右图所示。



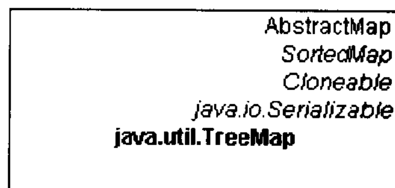
子类型的关系是传递性的：类型甲是类型乙的子类型，类型乙是类型丙的子类型，那么类型甲就是类型丙的子类型。

混合类型（Mixin Type）

如果一个类已经有一个主要的超类型，那么通过实现一个接口，这个类可以拥有另一个次要的超类型。这种次要的超类型就叫做混合类型（Mixin Type）。

如前面所谈到的，当一个具体类处于一个类的等级结构之中的时候，为这个具体类定义一个混合类型是可以保证基于这个类型的可插入性的关键。因此，混合类型是一个非常重要的概念。

显然，Java 接口是实现混合类型的最理想的工具，java.util.TreeMap 类的类图如右图所示。



由上图可知，TreeMap 类具有多个类型。它的主要类型是 AbstractMap，这是一种 Java 聚集；而 Cloneable 接口则给出一个次要类型，这个类型说明这个类的实例是可以安全地克隆的；同样，Serializable 也是一个次要类型，它表明这个类的实例是可以串行化的；而 SortedMap 则表明这个聚集类是可以排序的。

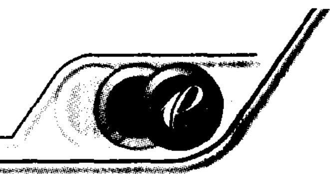
5.3 Java 接口常见的用法

Java 接口还有其他一些常见的使用方法。

单方法接口

顾名思义，一个单方法接口只含有一个方法。既然这个方法是这个接口的惟一内容，那么这个方法也就是单方法接口的核心。这种做法可以看做是 C 语言中函数指针的对应物。

一个函数指针允许一个 C 语言程序有存储和转移调用某一个函数的能力，一个单方



法接口的用意也很相似。在 Java 语言中有很多的单方法接口，比如，Runnable 接口要求实现一个没有参量和返还值的 run()方法。如代码清单 2 所示。

代码清单 2：实现单方法接口的源代码

```
public class MyThreadedClass extends SomeClass implements Runnable
{
   .
   .
   .
   public void run()
   {
       //write your code here
   }
}
```

再比如，事件监听接口 ActionListener 也规定了一个 actionPerformed()方法，如代码清单 3 所示。

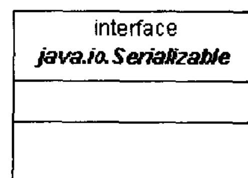
代码清单 3：ActionListener 接口的源代码

```
public interface ActionListener
{
    public abstract void actionPerformed(ActionEvent event);
}
```

标识接口

标识接口是没有任何方法和属性的接口。标识接口不对实现它的类有任何语义上的要求，它仅仅表明实现它的类属于一个特定的类型。

标识接口在 Java 语言中有一些很著名的应用，比如 java.io.Serializable 和 java.rmi.Remote 等接口便是标识接口，java.io.Serializable 接口如右图所示。

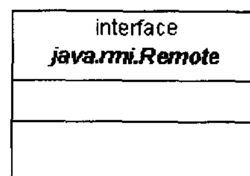


这个接口的源代码如代码清单 4 所示。

代码清单 4：java.io.Serializable 接口的源代码

```
package java.io;
public interface Serializable {}
```

另一个例子就是 RMI 库中的 Remote 接口，java.rmi.Remote 接口便是一个标识接口，如右图所示。



这个接口的源代码如代码清单 5 所示。

代码清单 5：java.io.Serializable 接口的源代码

```
package java.rmi;
public interface Remote {}
```

标识接口通常使用在工具类中，很少在其他地方使用。虽然在有些地方使用标识接口是很巧妙的做法，但是这个做法不可以滥用。一般而言，本书不推荐过多地使用这种做法。

常量接口

前面讲过，Java 接口可以用来声明一个新的类型。实际上，Java 接口应当仅仅用来声明一个新的类型，而不应当用来干任何别的事情。

读者可能会想，一个接口还能用来干什么呢？事实上，有一些“相当有经验”的 Java 程序员有一个使用 Java 接口的“绝招”：常量接口。所谓常量接口，是指用 Java 接口来声明一些常量，然后由实现这个接口的类使用这些常量。如下面的代码清单 6 所示。

代码清单 6：AppConstants 接口的源代码

```
public interface AppConstants
{
    public static final DATA_SOURCE_NAME = "WEBSPHERE_DATA_SOURCE";
    public static final USER_NAME = "APP_USER";
     public static final PASSWORD = "w1nter";
}
```

这样一来，凡是实现这个接口的 Java 类都会自动继承这些常量，并且都可以像使用自己的常量一样使用这些常量，而不必加上 AppConstants 的前缀。如果这些常量是在一个具体类中给出的，那么当然也是可以的，只是在应用的时候要加上类名作为前缀。这一做法被[TROST98]和[GRAND98-2]列为代码模式。

这种使用 Java 接口的方式是错误的[BLOCH01]。

读者可能会惊讶地发现，原来 Java 语言自身的 API 也有这样的错误做法。下图所示就是三处 Java 语言本身的 API 使用了常量接口的地方。

interface
java.io.ObjectStreamConstants
+STREAM_MAGIC:short
+STREAM_VERSION:short
+TC_BASE:byte
+TC_NULL:byte
+TC_REFERENCE:byte
+TC_CLASSDESC:byte
+TC_OBJECT:byte
+TC_STRING:byte
+TC_ARRAY:byte
+TC_CLASS:byte
+TC_BLOCKDATA:byte
+TC_ENDBLOCKDATA:byte

interface
java.util.zip.ZipConstants
+LOCSIG:long
+EXTSIG:long
+CENSIG:long
+ENDSIG:long
+LOCHDR:int
+EXTHDR:int
+CENHDR:int
+ENDHDR:int
+LOCVER:int
+LOCFLG:int
+LOCHOW:int
+LOCTIM:int

java.awt.dnd.DnDConstants
+ACTION_NONE:int
+ACTION_COPY:int
+ACTION_MOVE:int
+ACTION_COPY_OR_MOVE:int
+ACTION_LINK:int
+ACTION_REFERENCE:int

其中 ObjectStreamConstants 和 DnDConstants 是公开接口。由于 Java 语言给出了这两个公开接口以及公开的常量属性，Java 语言就只好永远将它们保留在自己的 API 中，即使它已经不再使用这些常量了。

如果 Java 语言从某一个版本开始，将这些常量接口删除，会怎样呢？结果是会有一大批客户程序停止工作。记住，任何公开的接口都是对客户的承诺，承诺一旦做出，就无

法撤回。

本书建议读者不要模仿这种常量接口的做法。

问答题

1．电视连续剧《日落紫禁城》将奸臣们的嘴脸形容为缝衣针："头尖身细白如银，上秤没有半毫分；眼睛长在屁股上，只认衣衫不认人"。请利用这个缝衣针嘲讽诗讲解一下接口的用处。

2．"生、旦、净、丑"是京剧的四个行当。所有的角色都属于一个行当。行当使京剧里的角色进一步抽象，因此，如果角色用类描述的话，接口恰好可以用来描述行当。比如《武松打虎》打虎中武松的行当便是武生，《凤还巢》里程雪娥的行当便是花旦等。请用继承的语言说明这种做法。

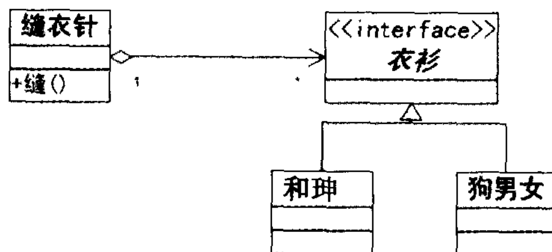
3．Java 构件模型的顶端是哪一个类型？

4．Java 语言是类型安全的吗？

5．Java 语言是怎样做到类型安全的呢？

问答题答案

1．接口在这里代表一种身份。缝衣针只认识接口，因此，不论是"和珅"还是"狗男女"，只要实现此接口，缝衣针便可以工作，如下图所示。



缝衣针的示范性代码如代码清单 7 所示。

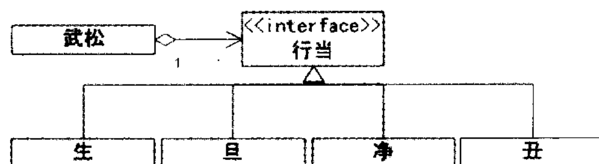
代码清单 7：缝衣针的源代码

```
public interface 缝衣针
{
    public void 缝(衣衫 x)
    {
        //Write code here
    }
}
```

2．"生、旦、净、丑"作为京剧的四个行当，是京剧所有的角色的抽象化。因此，如果角色用类描述的话，行当恰好可以用 Java 接口来描述。

比如《武松打虎》中武松的行当便是武生，《凤还巢》里程雪娥的行当便是花旦等。

下面所示的类图描述的正是京剧角色怎样持有“行当”属性的说明。



3. Java 构件模型的顶端是 java.lang.Object。所有的 Java 构件，包括数组对象，都以这个类为超类。



这是一个具体类，既不是 Java 接口也不是 Java 抽象类。

4. Java 是强类型的语言。这意味着 Java 编译器会对代码进行检查，以确定每一次赋值、每一次方法的调用是符合类型的。如果有任何不相符合的情况，Java 编译器就会给出错误。

类型检查是基于这样一个简单的事实：每一变量的声明都给这个变量一个类型；每一个方法（和构造子）的声明都给出这个方法的特征。这样一来，Java 编译器可以对任何的表达式推断出一个明显类型（Apparent Type），Java 编译器可以基于明显类型对类型进行检查。

Java 语言是类型安全的。这就是说，任何被 Java 编译器接收的合法的 Java 类保证是类型安全的。换言之，在程序运行期间，不会有任何类型的错误。一个 Java 程序根本不可能将一个本来属于一种类型的变量当做另一种类型处理，因此也就不会产生由此而引起的错误。

5. 简单地讲，Java 语言依靠三种机制做到了这一点：编译期间的类型检查，自动的存储管理，数组的边界检查。

参考文献

[COAD99] Peter Coad，Mark Mayfield，Jon Kern. Java Design，Building Better Apps & Applets. published by Prentice Hall，1999

[DLEA99] Doug Lea. Concurrent Programming in Java: Design Principles and Patterns. (ISBN 0-201-31009-0) Second Edition Published by Addison-Wesley，November 1999

[BLOCH01] Joshua Bloch. Effective Java – Programming Language Guide. published by Addison–Wesley，2001

[TROST98] Bill Trost. Define Constants in Interfaces. http://c2.com/cgi/wiki?DefineConstantsInInterfaces，1998

[JLS00] James Gosling，Bill Joy，Guy Steele，Gilad Bracha. The Java Language Specification. Second Edition，Addison–Wesley，Boston，2000

第 6 章　专题：抽象类

在 Java 语言里，类有两种：一种是具体类，另一种是抽象类。具体类可以实例化，抽象类不可以实例化。

6.1　什么是抽象类

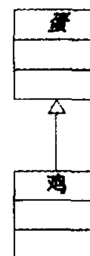
抽象类仅提供一个类型的部分实现。抽象类可以有实例变量，以及一个或多个构造子。抽象类可以同时有抽象方法和具体方法。

一个抽象类不会有实例，这些构造子不能被客户端调用来创建实例。一个抽象类的构造子可以被其子类调用，从而使一个抽象类的所有子类都可以有一些共有的实现，而不同的子类可以在此基础上有其自己的实现。

抽象类和子类的这种关系实际上是模版方法模式的应用。

先有蛋，后有鸡

套用“先有鸡，还是先有蛋”的争论，假设蛋代表抽象，而鸡代表具体。蛋决定鸡的性状，鸡是蛋的“实现”，蛋的类型决定了鸡的类型。从这个角度来讲，先有蛋，后有鸡，如右图所示。



Java 抽象类与 Java 接口一样，都用来声明一个新的类型，并且作为一个类型的等级结构的起点。但是，Java 接口具有比 Java 抽象类更好的特性，因此，应当优先使用 Java 接口声明一个超类型。

6.2　抽象类的用途

抽象类通常代表一个抽象概念，它提供一个继承的出发点。而具体类则不同，具体类可以实例化，应当给出一个有商业逻辑实现的对象模版。由于抽象类不可以实例化，因此一个设计师设计一个新的抽象类，一定是用来继承的。

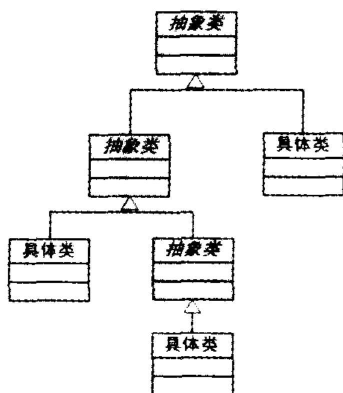
而这一个声明倒过来也是对的：具体类不是用来继承的。

具体类不是用来继承的

Scott Meyers[MEYERS96]曾指出，只要有可能，不要从具体类继承。

如下图所示，此类图就给出了一个继承形成的等级结构的典型例子。可以看出，所有

的继承都是从抽象类开始的，而所有的具体类都没有子类。



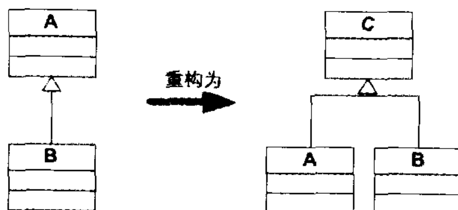
换言之，在一个以继承关系形成的等级结构里面，树叶节点均应当是具体类，而树枝节点均应当是抽象类（或者 Java 接口）。

这样的设计是所有的 Java 设计师都应当努力做到的。

代码重构的建议

如果在一个原始的设计里，有两个具体类之间有继承关系，那么最可能的修改方案是怎样的呢？

假设有两个具体类，类 A 和类 B，类 B 是类 A 的子类，那么一个最简单的修改方案应当是建立一个抽象类（或者 Java 接口）C，然后让类 A 和类 B 成为抽象类 C 的子类，如下图所示。



上面所给出的代码重构的例子实际上具有更加广泛的意义，这就是里氏代换原则。这一原则会在本书的“里氏代换原则（LSP）”一章中加以介绍。

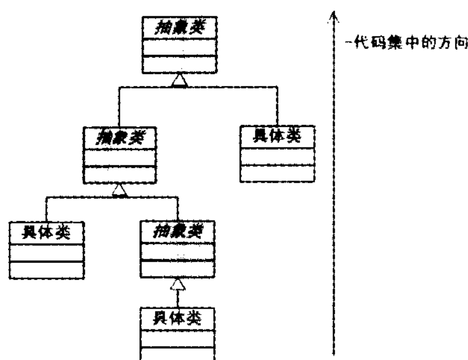
当然，在这一重构之后要做的，便是如何处理类 A 和类 B 的共同代码和共同数据，这就是本章要引入的下面两小节要讲解的内容。

抽象类应当拥有尽可能多的共同代码

在一个从抽象类到多个具体类的继承关系中，共同的代码应当尽量移动到抽象类里。

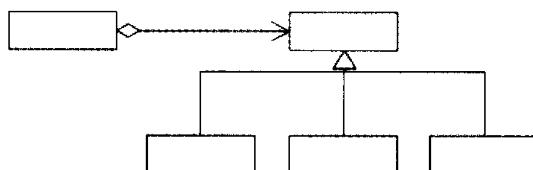
在一个继承的等级结构中，共同的代码应当尽量向等级结构的上方移动，如下图所示。把重复的代码从子类里面移动到超类里面，可以提高代码的复用率。由于代码在共同的超类而不是几个子类中出现，在代码发生改变时，设计师只需要修改一个地方。这对代码的

复用明显是有利的。



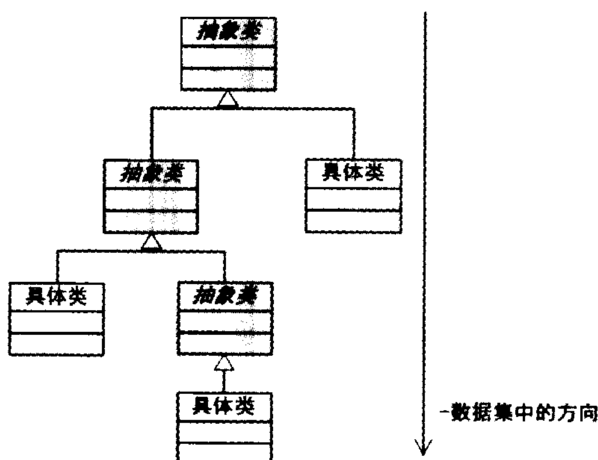
一个对象从超类继承而来的代码，在不使用时不会造成对资源的浪费。回到前面所讨论的代码重构的例子，设计师应当将类 A 和类 B 的共同代码尽量移动到抽象超类 C 里面，这样可以保证最大限度的复用。

一个典型的例子就是策略模式。在策略模式中，抽象策略角色的分量越重越好，也就是说尽可能将公共的方法移动到抽象策略角色中，如下图所示。



抽象类应当拥有尽可能少的数据

与代码的移动方向相反的是，数据的移动方向是从抽象类到具体类，也即从继承的等级结构的高端向等级结构的低端移动，如下图所示。一个对象的数据不论是否使用都会占用资源，因此数据应当尽量放到具体类或者等级结构的低端。



回到前面所讨论的代码重构的例子，设计师应当将类 A 和类 B 的数据保持在各自的类中，而不是移动到抽象超类 C 里面，这样可以保证节省内存资源。

6.3 基于抽象类的模式和原则

抽象类的使用不是简单的几句话可以概括的，下面几条是应当注意的设计思想。

针对抽象编程

针对抽象编程，不要针对具体编程。这就是依赖倒转原则（Inverse Dependency Principle）所讲的内容，请参见本书的“依赖倒转原则（DIP）”一章。

换言之，应当针对抽象类编程，不要针对具体子类编程，这一原则点出了抽象类对代码复用的一个最重要的作用。

以“鸡和蛋”的例子解释，就是要把编程针对到“蛋”上，而不是“鸡”身上。

正确使用继承

在 Java 语言中，继承关系可以分成两种：一种是类对接口的实现，称做接口继承；另一种是类对类的继承，称做实现继承。第二种继承关系是很容易被滥用的一种复用工具。

正如本章前面所指出的，抽象类是用来继承的，因此抽象类注定要与继承关联在一起。只要可能，尽量使用合成（Composition），而不要使用继承来达到复用的目的。关于这一点，请见本书“组合/聚合复用原则（CARP）”一章。

关于恰当使用继承作为复用的工具的讨论，请见本书的“里氏代换原则（LSP）”、“合成/聚合复用原则（CARP）”以及“模版方法（Template Method）模式”等章节。

模版方法模式

抽象类在本书所讨论的各个模式里面扮演了重要的角色。几乎所有的模式都涉及到抽象类，但是模版方法模式不仅仅使用抽象类和继承关系作为对模式所涉及的角色的抽象化，模版方法模式根本就是关于继承的模式。关于模版方法模式，请见本书“模版方法（Template Method）模式”一章。

当读者阅读完本书所有的章节后，会对抽象类的使用有一个新的高度上的理解。

6.4 什么时候才应当使用继承复用

Peter Coad 认为，继承代表“一般化/特殊化”关系，其中基类代表一般，而衍生类代表特殊，衍生类将基类特殊化或扩展化。只有当以下的 Coad 条件全部被满足时，才应当



使用继承关系：

（1）子类是超类的一个特殊种类，而不是超类的一个角色，也就是要区分“Has-A”与“Is-A”两种关系的不同。Has-A 关系应当使用聚合关系描述，而只有 Is-A 关系才符合继承关系。

（2）永远不会出现需要将子类换成另一个类的子类的情况。如果设计师不是很肯定一个类会不会在将来变成另一个类的子类的话，就不应当将这个类设计成当前这个超类的子类。

（3）子类具有扩展超类的责任，而不是具有置换掉（Override）或注销掉（Nullify）超类的责任。如果子类需要大量地置换掉超类的行为，那么这个子类不应当成为这个超类的子类。

（4）只有在分类学角度上有意义时，才可以使用继承，不要从工具类继承。

里氏代换原则是可否使用继承关系的准绳。上面的 Coad 条件以比里氏代换原则更加通俗易懂的方式讲解了继承的使用，Coad 条件的表达不如里氏代换原则严格，但是一般而言，凡是不符合 Coad 条件的均不会满足里氏代换原则。Coad 条件可以作为里氏代换原则的衍生物。

下面就逐条讨论一下 Coad 的原则。

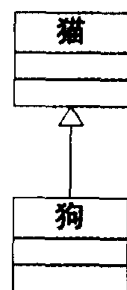
区分“Has-A”与“Is-A”

当一个类是另一个类的角色时，不应当使用继承描述这种关系。如果仔细考察就会发现，这种情况一定不满足里氏代换原则。

详细的讨论请见本书的“合成/聚合复用原则（CARP）”一章的相关讨论。

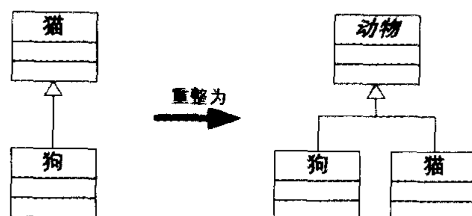
子类扩展超类的责任

子类应当扩展超类的责任，而不是置换掉（Override）或撤销掉（Nullify）超类的责任。如果一个子类需要将继承自超类的责任取消或置换后才能使用的话，很有可能这个子类根本就不是哪个超类的子类。



将狗设计成猫的子类，如右图所示。猫有上树的能力，狗没有。为了继承关系成立，只好将猫上树的能力取消掉，这个继承关系显然是错误的。

正确的继承关系是引入一个抽象类，在这里就是“动物”类，将两个具体类设计成抽象类的子类，如下图所示。



一般而言，如果子类需要置换掉太多的超类的行为，那么一定是因为子类的行为与超

类有太大的区别。这个时候，很有可能子类并不能取代超类出现在任何需要超类的地方，也就是说它们不满足里氏代换原则。

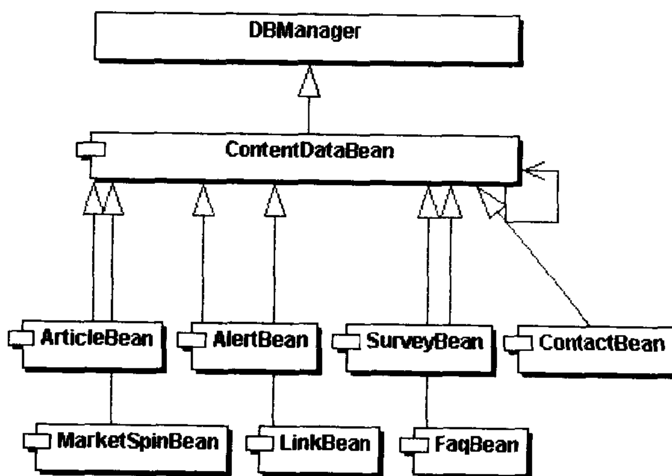
不要从工具类继承

只有当有分类学意义时，才使用继承。一般而言，一个商业类型不会与一个工具类型有这种分类学上有意义的关系，因此不要从工具类继承。从里氏代换原则的角度上考虑，一个工具类必定有自己的行为特性，一个商业类型封装了商业逻辑，不可能是工具类型的一种。换言之，几乎没有例外，从工具类型继承是错误的。

下面是一个真实的华尔街金融网站的系统设计，是一个动态的内容网站，设计师将内容划分为 Article、Alert、Survey、Contact、MarketSpin、Link 和 Faq 等几种。所有这些内容都分别由一个 JavaBean 负责，而所有的这些 JavaBean 都是一个 ContentDataBean 类的子类。

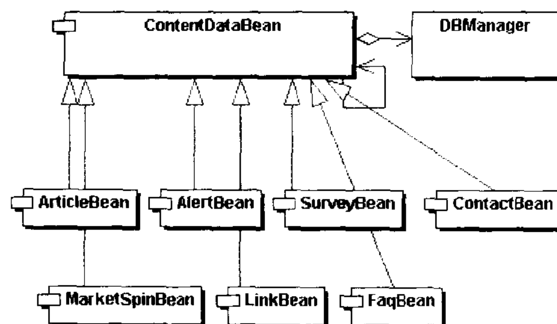
同时，由于系统是由数据库驱动的，因此系统需要一个 DBManager 的工具类，提供所有的数据库功能，比如 executeSQL()，openQuery()等方法。

如果到此为止，一切都还算好。但是负责这个设计的设计师灵机一动，将 ContentDataBean 设计成了 DBManager 类的子类，这样的好处不是很明显嘛，所有的子类一下子就得到了所有 DBManager 的数据库功能，如下图所示。



这就是将继承关系当成了权宜之计，而没有做分类学上的考虑。所有的 ContentDataBean 的各个内容子类都是同一类型的东西，而这些内容类和 DBManager 的区别就像猫和狗的区别一样大。这是一种很常见的滥用继承关系的例子。

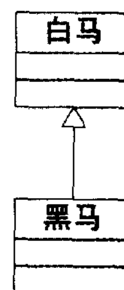
当然这种滥用的情况不能使用引进一个抽象类的办法重构纠正，而是应当采取将继承关系改为委派关系的办法纠正。也就是说，将从 DBManager 到 ContentDataBean 的继承关系改成从 ContentDataBean 到 DBManager 的委派关系，如下图所示。



问答题

1. 请使用“种瓜得瓜、种豆得豆”的古谚来说明抽象和继承的概念。

2. David 在考察黑马和白马的关系。David 认为黑马和白马都可以用具体 Java 类代表，黑马可以看做是白马的一些变化，也就是说，把白马涂黑就得到了黑马，如右图所示。请问 David 的这个看法对吗？

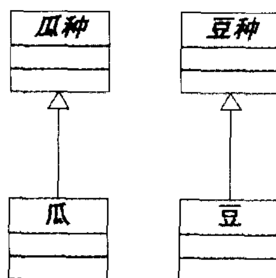


3. 请使用抽象和继承的概念重新诠释“种下龙种，收获跳蚤”的意义。

问答题答案

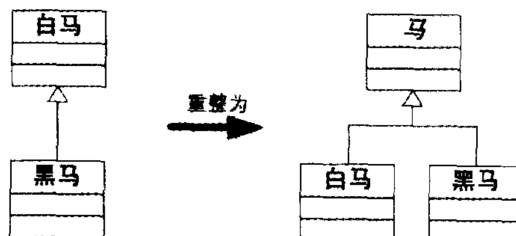
1. “种瓜得瓜、种豆得豆”的古谚讲的是瓜的种子结出瓜，而豆的种子结出豆。播下瓜种不可能长出豆，播下豆种不可能长出瓜。

植物的种子是植物特性的抽象化，植物则是其种子的具体化。播下某一种植物的种子，只可能得到这种植物，而不可能得到另一种植物，如下图所示。

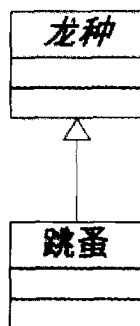


2. David 的看法是错误的。

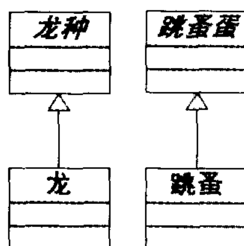
从具体类继承是不妥当的。一个重构的方案是引入一个抽象类，作为两个具体类的超类，如下图所示。



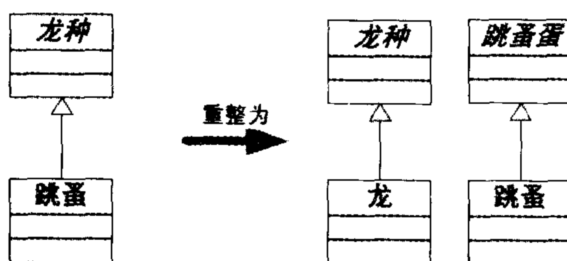
3. 从抽象和继承的概念上讲，"播下龙种，收获跳蚤"讲的是错误地从一个"龙种"抽象类得出一个"跳蚤"具体类，如下图所示。



"龙种"的子类应当是"龙子、龙孙"，而"跳蚤"的抽象超类应当是"跳蚤蛋"，如下图所示。



因此，从代码重构的角度上讲，应当把跳蚤和龙种的关系调整过来，如下图所示。



这是一个荒诞的例子，但是在有经验的 OO 设计师看来，很多系统的设计都和"播下龙种，收获跳蚤"一样荒诞，只是设计师本人没有察觉而已。

参考文献

[MEYERS96] Scott Meyers. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Addision-Wesley, 1996

[COAD91] P. Coad, E. Yourdon. Object–oriented Analysis. Prentice-Hall Inc, USA, 1991

[COAD99] Peter Coad, Mark Mayfield, Jon Kern. Java Design, Building Better Apps & Applets. published by Prentice Hall, 1999

第 7 章　里氏代换原则（LSP）

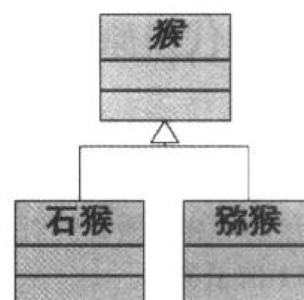
从“开-闭”原则中可以看出面向对象设计的重要原则是创建抽象化，并且从抽象化导出具体化。具体化可以给出不同的版本，每一个版本都给出不同的实现。

从抽象化到具体化的导出要使用继承关系和这里要引入的里氏代换原则（Liskov Substitution Principle，常缩写为 LSP）。里氏代换原则由 Barbara Liskov 提出。

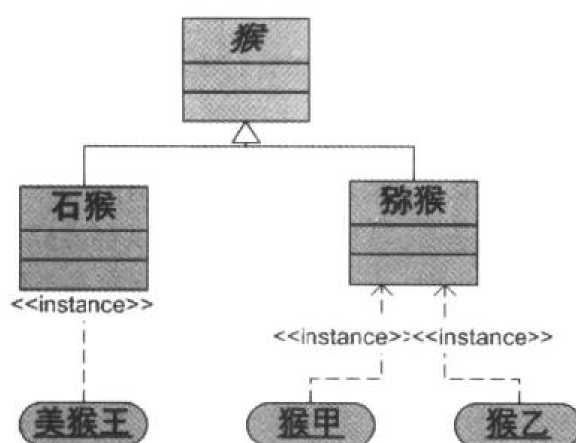
在仔细研究里氏代换原则之前，先看一看美猴王的智慧可以带给我们什么样的启发。

7.1　美猴王的智慧

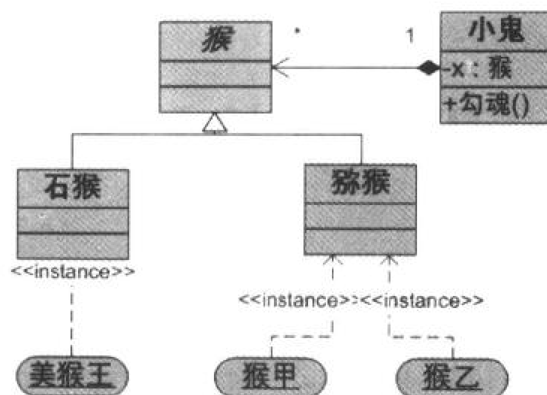
当年美猴王打到幽冥界，在生死簿寻找自己的姓名“直到那魂字一千三百五十号上，方注着孙悟空名字，乃天产石猴，该寿三百四十二岁……（悟空）饱掭浓墨……把猴属之类，但有名者，一概勾之……自此，山猴都有不老者，以阴司无名故也”。猴类是基类，石猴、猕猴是子类，子类的颜色与基类的颜色不同，如右图所示。



显然，幽冥地府管理一切生灵的生死的方法是以类来区分的。美猴王是一个石猴，而石猴类与猕猴类一样，都是猴类的子类，如下图所示。



因此，美猴王将猴类有姓名者统统消掉，显然是因为阴曹地府和勾魂的小鬼并不区分石猴类与猕猴类，如下图所示。



换言之，猴类适用的，猕猴和石猴类全都适用，这其实就是里氏代换原则。

7.2 什么是里氏代换原则

里氏代换原则

里氏代换原则的严格表达是：

如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有变化，那么类型 T2 是类型 T1 的子类型。

换言之，一个软件实体如果使用的是一个基类的话，那么一定适用于其子类，而且它根本不能察觉出基类对象和子类对象的区别。

比如，假设有两个类，一个是 Base 类，另一个是 Derived 类，并且 Derived 类是 Base 类的子类。那么一个方法如果可以接受一个基类对象 b 的话：

```
method1(Base b)
```

那么它必然可以接受一个子类对象 d，也即可以有 method1(d)。

里氏代换原则是继承复用的基石。只有当衍生类可以替换掉基类，软件单位的功能不会受到影响时，基类才能真正被复用，而衍生类也才能够在基类的基础上增加新的行为。

反过来的代换不成立

必须指出的是，反过来的代换则不成立，即如果一个软件实体使用的是一个子类的话，那么它不一定适用于基类。如果一个方法 method2 接受子类对象为参数的话：

```
method2(Derived d)
```

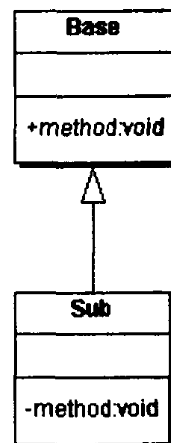
那么一般而言不可以有 method2(b)。

Java 语言对里氏代换的支持

在编译时期，Java 语言编译器会检查一个程序是否符合里氏代换，当然这是一个无关实现的、纯语法意义上的检查。

里氏代换要求凡是基类型使用的地方，子类型一定适用，因此子类必须具备基类型的全部接口。或者说，子类型的接口必须包括全部的基类型的接口，而且还有可能更宽。如果一个 Java 程序破坏这一条件，Java 编译器就会给出编译时期错误。

举例而言，一个基类 Base 声明了一个 public 方法 method()，那么其子类型 Sub 可否将这个方法的访问权限从 public 改换成为 package-private 呢？换言之，子类型可否使用一个低访问权限的方法 private method()将超类型的方法 public method()置换（override）掉呢？　超类 Base 与子类 Sub 的结构图如右图所示。



从里氏代换的角度考察这个问题，就不难得出答案，因为客户端完全有可能调用超类型的公开方法。如果以子类型代之，这个方法却变成了私有的，客户端不能调用。显然这是违反里氏代换法则的，Java 编译器根本就不会让这样的程序过关。

Java 语言对里氏代换支持的局限

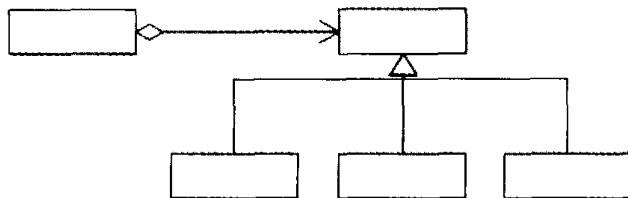
Java 编译器的检查是有局限的。为什么呢？举例来说，描写一个物体大小的量有精度和准确度两种属性。所谓精度，就是这个量的有效数字有多少位；而所谓准确度，是这个量与真实的物体大小相符合到什么程度。一个量可以有很高的精度，但是却无法与真实物体的情况相吻合。Java 语言编译器所能够检查的，仅仅是相当于精度的属性而已，它无法检查这个量与真实物体的差距。

换言之，Java 编译器不能检查一个系统在实现和商业逻辑上是否满足里氏代换法则。一个著名的例子，就是“正方形类是否是长方形类的子类”的问题。本章会在后面详细讨论这种具体实现和商业逻辑上的问题。

7.3　里氏代换原则在设计模式中的体现

策略模式

关于策略模式，请读者阅读本书的“策略（Strategy）模式”一章。策略模式是所有的设计模式中比较易于理解的模式。策略模式讲的是：“如果有一组算法，那么就将每一个算法封装起来，使得它们可以互换。”策略模式的简略类图如下图所示。



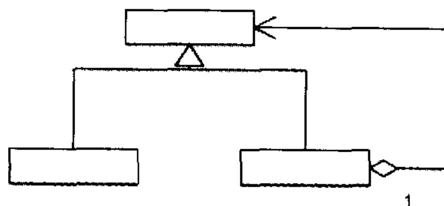
封装的概念是不难理解的，而使得所有的算法可以互换，则需要将所有的具体策略角色放到一个类型等级结构中，使它们拥有共同的接口。显然这种互换性依赖的是对里氏代换原则的遵守：

```
AbstractStrategy s = new ConcreteStrategyA();
```

从上面的代码片断可以看出，客户端依赖于基类类型，而变量的真实类型则是具体策略类。这是具体策略角色可以"即插即用（Pluggable）"的关键。

合成模式

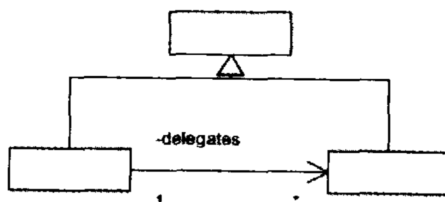
合成模式通过使用树结构描述整体与部分的关系，从而可以将单纯元素与复合元素同等看待。由于单纯元素和复合元素都是抽象元素角色的子类，因此两者都可以替代抽象元素出现在任何地方。合成模式的简略类图如下图所示。



显然，里氏代换法则是合成模式能够成立的基础。关于合成模式，请参见本书的"合成（Composite）模式"一章。

代理模式

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。代理模式能够成立的关键，就在于代理模式与真实主题模式都是抽象主题角色的子类。客户端只知道抽象主题，而代理主题可以替代抽象主题出现在任何需要的地方，而将真实主题隐藏在幕后。代理模式的简略类图如下图所示。



显然，里氏代换原则是代理模式能够成立的基础。关于代理模式，请参见本书的"代

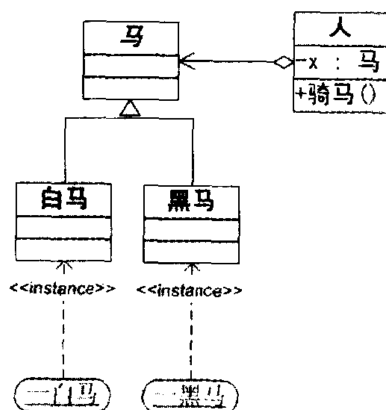
理（Proxy）模式”一章。

7.4 墨子论“取譬”

《墨子》有大取和小取两章，“取”是“取譬”的意思。使用面向对象的语言来解释，“取譬”研究的就是类和类的实例。

白马与马

《墨子·小取》中说，“白马，马也；乘白马，乘马也。骊马，马也；乘骊马，乘马也。”所谓骊马，便是黑色的马。墨子在这里说的是，不论黑马、白马均是马的一种。既然马可以骑，那么白马和黑马必可骑，如下图所示。



“马”便是抽象的马，而白马和黑马则是马的具体子类。一白马则是“白马”类的实例；一黑马则是“黑马”类的实例。如果一个操作适用于马，那么必然适用于白马和黑马。换言之，如果有一个方法 rideHorse()接受 Horse 作为参量的话：

```
rideHorse(Horse aHorse);
```

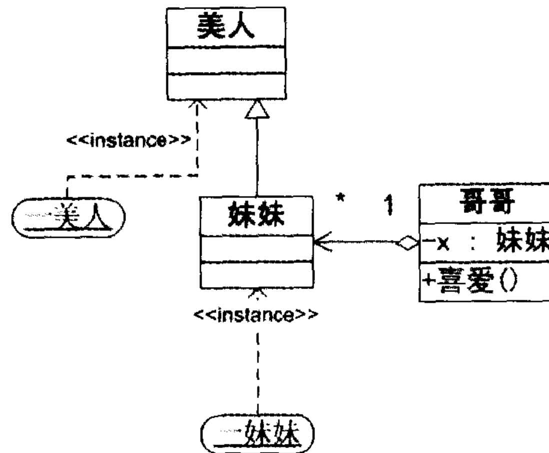
那么 rideHorse(aWhiteHorse)和 rideHorse(aBlackHorse)均是成立的，也就是说，rideHorse()也可以接受 aWhiteHorse 和 aBlackHorse 作为参数。

这也就道出了里氏代换的精髓。

反过来的代换不成立

应当指出的是，墨子同时指出了反过来并不成立。《墨子·小取》说：“娣，美人也，爱娣，非爱美人也……盗，人也；恶盗，非恶人也。”娣便是妹妹，哥哥喜爱妹妹，是因为两人的兄妹关系，而不是因为妹妹是个美人。因此，喜爱妹妹不代表喜爱美人。使用面向对象的语言来说，美人是一个基类，妹妹是美人的子类。哥哥作为一个类有个“喜爱()”方法，接受妹妹作为参量。那么，这个“喜爱()”方法一般不能接受美人类的实例，如下

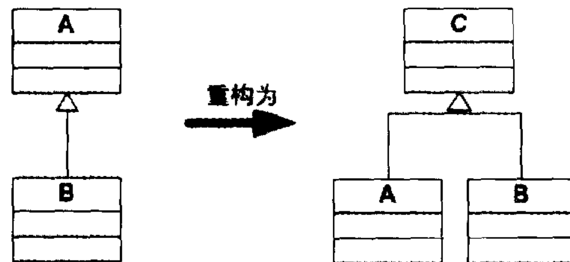
图所示。



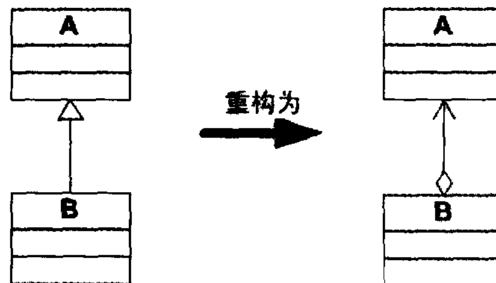
7.5 从代码重构的角度理解

里氏代换原则讲的是基类与子类的关系。只有当这种关系存在时，里氏代换关系才存在；反之则不存在。如果有两个具体类 A 和 B 之间的关系违反了里氏代换原则的设计，根据具体情况可以在下面的两种重构方案中选择一种：

（1）创建一个新的抽象类 C，作为两个具体类的超类，将 A 和 B 的共同行为移动到 C 中，从而解决 A 和 B 行为不完全一致的问题，如下图所示。



（2）从 B 到 A 的继承关系改写为委派关系，如下图所示。



本章下面以一个例子来说明第一种重构方案，而关于第二种重构方案，请参见本书的

“合成/聚合复用原则（CARP）”一章。

长方形和正方形

在西方，墨子的马与白马的思辨大多数人都不知道，这里给出的是在西方很著名的思辨，也就是正方形是否是长方形的子类的问题（与此相类似的，还有圆形与椭圆形的问题）。

一个长方形 Rectangle 类的定义如代码清单 1 所示。

代码清单 1：Rectangle 类的源代码

```
package com.javapatterns.liskov.version1;
public class Rectangle
{
    private long width;
    private long height;

    public void setWidth(long width)
    {
        this.width = width;
    }

    public long getWidth()
    {
        return this.width;
    }

    public void setHeight(long height)
    {
        this.height = height;
    }

    public long getHeight()
    {
        return this.height;
    }
}
```

当 width 与 height 相等时，就得到了正方形对象。因此，长方形的对象中有一些是正方形对象。考虑 Square 类，其源代码如代码清单 2 所示。

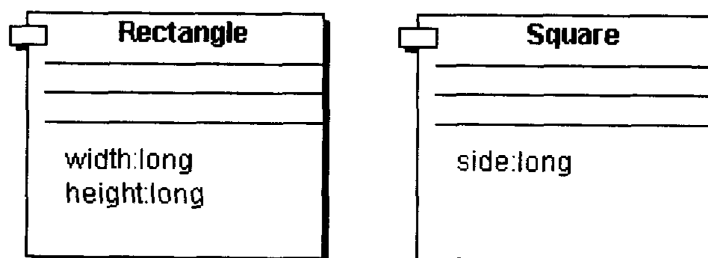
代码清单 2：Square 类的源代码

```
package com.javapatterns.liskov.version1;
public class Square
{
    private long side;

    public void setSide(long side)
```

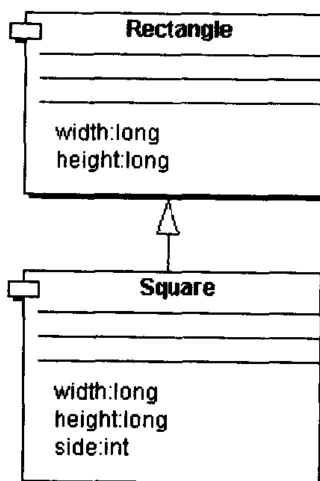
```
    {
        this.side = side;
    }
    public long getSide()
    {
        return side;
    }
}
```

因为这个正方形类不是长方形类的子类，而且也不可能成为长方形类的子类，因此在 Rectangle 类与 Square 类之间不存在里氏代换关系，如下图所示。



正方形不可以作为长方形的子类

读者可能会问，为什么正方形不可以成为长方形的子类呢？下面所示的类图就将正方形设置成了长方形的子类。



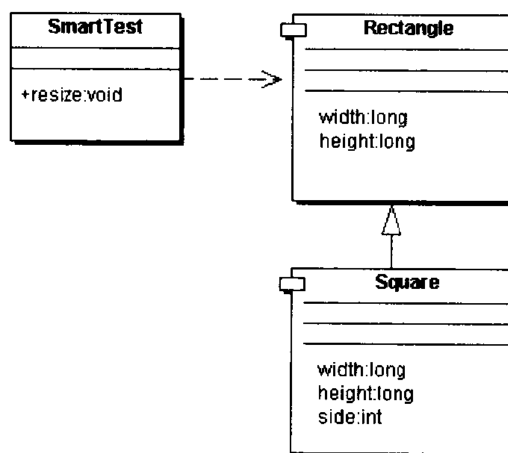
为什么不能在内部将 height 设成 width 呢？Square 类的源代码如代码清单 3 所示。

代码清单 3：新的 Square 类的源代码

```
package com.javapatterns.liskov.version2;
public class Square extends Rectangle
{
```

```
    private int side;
    public void setWidth(long width)
    {
        setSide(width);
    }
    public long getWidth()
    {
        return getSide();
    }
    public void setHeight(long height)
    {
        setSide(height);
    }
    public long getHeight()
    {
        return getSide();
    }
    public int getSide()
    {
        return side;
    }
    public void setSide(int side)
    {
        this.side = side;
    }
}
```

这样，只要 width 或 height 被赋值，那么 width 和 height 会同时被赋值，从而使长方形的长和宽总是相等的。这看上去像是骗过了 Java 编译器，从而解决了问题。但是如果客户端使用一个 Square 对象调用下面的 resize()方法时，就会得出与使用一个 Rectangle 对象不同的结论。当传入的是一个 Rectangle 对象时，这个 resize()方法会将宽度不断增加，直到它超过长度才停下来。如果传入的是一个 Square 对象时，这个 resize()方法就会将正方形的边不断地增加下去，直到溢出为止，如下图所示。



SmartTest 类的源代码如代码清单 4 所示。

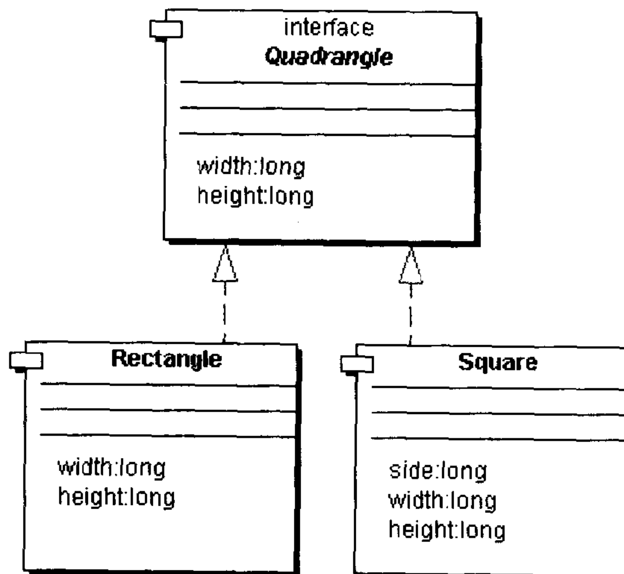
代码清单 4：SmartTest 类的源代码

```
package com.javapatterns.liskov.version2;
public class SmartTest
{
    public void resize(Rectangle r)
    {
        while (r.getHeight() <= r.getWidth() )
        {
            r.setWidth(r.getWidth() + 1);
        }
    }
}
```

换言之，里氏代换原则被破坏了，因此 Square 不应当成为 Rectangle 的子类。这个例子很重要，它意味着里氏代换与通常的数学法则和生活常识有不可混淆的区别。

代码的重构

根据墨子在两千多年前的思辨，白马与黑马同属于马。那么 Rectangle 与 Square 到底是怎样的关系呢？它们都应当属于四边形（Quadrangle）类的子类，如下图所示。



这样，通过发明一个 Quadrangle（四边形）类，并将 Rectangle 与 Square 变成它的具体子类，就解决了 Rectangle 与 Square 的关系不符合里氏代换原则的问题。例子所涉及的源代码如下。首先是作为 Java 接口的四边形类的源代码，如代码清单 5 所示。

代码清单 5：Quadrangle 类的源代码

```
package com.javapatterns.liskov.version3;
```



```
public interface Quadrangle
{
    public long getWidth();
    public long getHeight();
}
```

可以看出，这个四边形类只声明了两个取值方法，没有声明任何的赋值方法。长方形类的源代码如代码清单 6 所示，长方形是四边形的子类。

代码清单 6：Rectangle 类的源代码

```
package com.javapatterns.liskov.version3;
public class Rectangle implements Quadrangle
{
    private long width;
    private long height;

    public void setWidth(long width)
    {
        this.width = width;
    }

    public long getWidth()
    {
        return this.width;
    }

    public void setHeight(long height)
    {
        this.height = height;
    }

    public long getHeight()
    {
        return this.height;
    }
}
```

可以看出，长方形类确实有赋值方法。正方形类的源代码如代码清单 7 所示，它也给出了相应的赋值方法。

代码清单 7：Rectangle 类的源代码

```
package com.javapatterns.liskov.version3;
public class Square implements Quadrangle
{
    private long side;
    public void setSide(long side)
    {
```

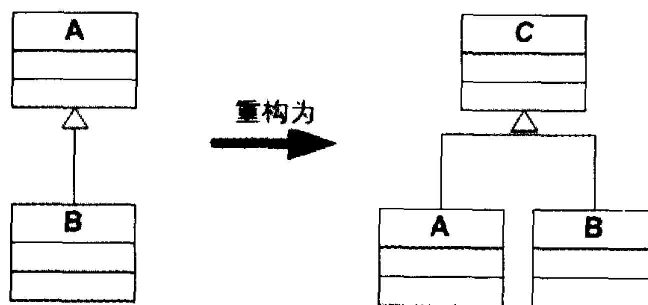
```
            this.side = side;
        }
        public long getSide()
        {
            return side;
        }
        public long getWidth()
        {
            return getSide();
        }
        public long getHeight()
        {
            return getSide();
        }
    }
```

那么破坏里氏代换的问题在这里是怎样避免的呢？秘密在于基类 Quadrangle 类没有赋值方法，因此类似于 SmartTest 的 resize()方法不可能适用于 Quadrangle 类型，而只能适用于不同的具体子类 Rectangle 和 Square，因此里氏代换原则不可能被破坏。

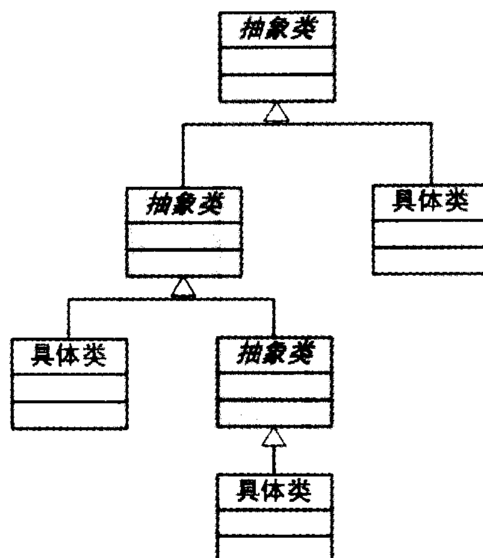
从抽象类继承

正如本书在“专题：抽象类”一章中所指出的，应当尽量从抽象类继承，而不从具体类继承。Quadrangle 接口相当于抽象类，使用这个抽象角色解决了两个具体类 Rectangle 和 Square 具有继承关系的问题。

一般而言，如果有两个具体类 A 和 B 有继承关系，如下图所示，那么一个最简单的修改方案应当是建立一个抽象类 C，然后让类 A 和类 B 成为抽象类 C 的子类，如下图所示。



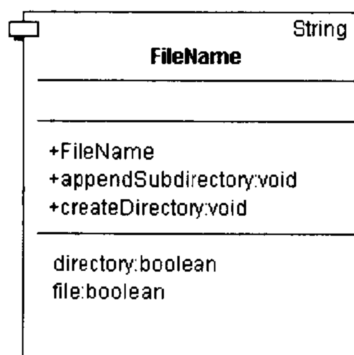
本章关于 Rectangle 和 Square 的讨论恰好是对这一重构方案的注解。如果有一个由继承关系形成的等级结构的话，那么在等级结构的树图上面所有的树叶节点都应当是具体类；而所有的树枝节点都应当是抽象类或者 Java 接口，如下图所示。



当然，这仅仅是一条指导性的原则，在使用时要针对具体情况做具体分析。详细的讨论请见本书的“专题：抽象类”一章。

问答题

1. David 是一个初级 Java 设计师，他很喜欢使用 X 语言，因为 X 语言允许他很容易地使用继承关系。这一次 David 需要一个类描述文件名，David 把这个类叫做 FileName。他认为 FileName 基本上是 X 语言提供的 String 类型的特殊情况，因此他认为应当设计一个 String 类的子类，然后加上一些 FileName 类所需要的方法，其设计图如下图所示。



请使用里氏代换原则分析一下这个做法是否正确。（同样，这也是一个很有名的思辨题目，原题是以 C++语言给出的）

2. 本章曾讲过，正方形不可以作为长方形的子类，因为正方形并不具有长方形所有的行为。请问不变正方形是否可以成为长方形的子类呢？

所谓不变正方形，就是边长不会发生变化的正方形，也就是遵守不变模式的正方形。关于不变模式请见本书的“不变模式”一章。

3. 请从里氏代换原则的角度考察 java.util 库中的 Properties 与 Hashtable 的关系是否

合适。

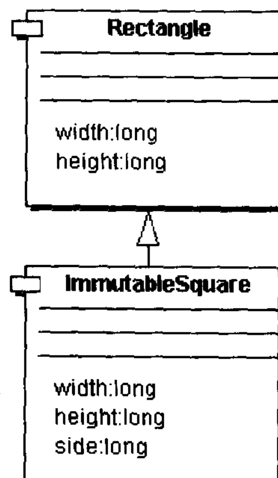
问答题答案

1．这样做是不符合里氏代换原则的。

里氏代换原则说，如果一个方法对一个基类成立的话，那么一定适用于其子类。任意两个 String 对象相加都可以给出一个新的有效 String 对象，而两个 FileName 对象相加却不一定给出一个新的有效 FileName 对象。换言之，FileName 对象并不能实现 String 对象的所有行为。

最后指出，由于 Java 语言中的 String 类型是一个 final 类型，因此不可以继承。读者可以想像所谓的 X 语言就是一个类似于 Java 语言的，但恰好具有一个不是 final 的 String 类型的语言。

2．不变正方形（Immutable Square）可以成为长方形的子类，因为不变正方形通过废弃边长变化的可能性，从而完全遵守了长方形的行为，如下图所示。



这个 ImmutableSquare 类的源代码如代码清单 8 所示。

代码清单 8：不变正方形的源代码

```
package com.javapatterns.liskov.version4;
public class ImmutableSquare extends Rectangle
{
    private long side;
    public void setWidth(long width)
    {
//        setSide(width);
    }
    public long getWidth()
    {
        return getSide();
    }
    public void setHeight(long height)
```

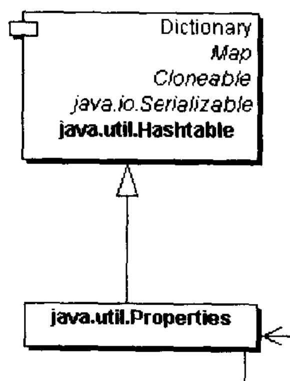
```
    {
//          setSide(height);
    }
    public long getHeight()
    {
        return getSide();
    }
    public long getSide()
    {
        return side;
    }
    public void setSide(long side)
    {
//          this.side = side;
    }
}
```

可以看出，不变正方形其实就是废弃了边长变化可能性的正方形。

读者可以通过设计一个类似于代码清单 3 所给出的 SmartTest 类的测试程序来测试不变正方形与长方形行为的符合性。

但是，由于继承应当用来扩展超类的功能，而不是置换（Override）或者撤销超类的功能，因此这里的做法虽然技术上并不违反里氏代换原则，却并不值得推荐。

3. 从里氏代换的角度来看，Properties 与 Hashtable 的关系是不合适的，如下图所示。



显然，Properties 是一种特殊的 Hashtable，它只接受 String 类型的键（Key）和值（Value）。但是，其超类型则可以接受任何类型的键和值。这就意味着，在一些需要非 String 类型的键和值的地方，Properties 不能够取代 Hashtable。

这是一个 Java 语言 API 本身违反里氏代换原则的反面教材。更加详细的讨论，请参见本书的“合成/聚合复用原则（CARP）”一章。

参考文献

[LISKOV94] Barbara Liskov and Jeanette Wing. A Behavioral Notion of Subtyping，ACM

Transactions on Programming Languages and Systems. Vol 16, No 6, November, 1994, pages 1811-1841

[MARTIN96b] R.C. Martin. The Liskov Substitution Principle——C++ Report. (1996, Mar)

[WORDEN00b]Brent Worden . The Liskov Substitution Principle . http://www.brent.worden.org

[WORDEN00b] Brandon Goldfedder. The Joy of Patterns - Using Patterns for Enterprise Development. Addison–Wesley, 2002

[MOZI00] 孙波（注释）. 墨子全文注释本. 华夏出版社, 2000

[KIRK02]Kirk Knoernschild. Java Design–Objects, UML, and Process. Addison-Wesley, 2002

第 8 章　依赖倒转原则（DIP）

实现“开–闭”原则的关键是抽象化，并且从抽象化导出具体化实现。如果说“开–闭”原则是面向对象设计的目标的话，依赖倒转原则就是这个面向对象设计的主要机制[MARTIN00]。

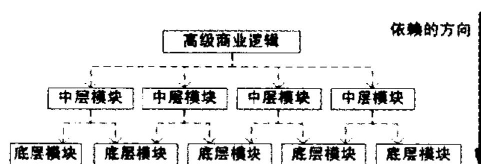
依赖倒转原则讲的是：要依赖于抽象，不要依赖于具体。

8.1　为何而“倒转”

为什么要使用“倒转（Inversion）”一词，依赖倒转（Dependence Inversion）的意义是什么？

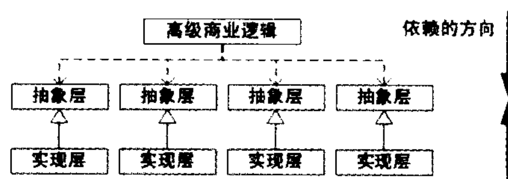
简单地说，传统的过程性系统的设计办法倾向于使高层次的模块依赖于低层次的模块；抽象层次依赖于具体层次。倒转原则是要把这个错误的依赖关系倒转过来，这就是“依赖倒转原则”的来由。

抽象层次依赖于具体层次的含义是什么呢？抽象层次包含的是应用系统的商务逻辑和宏观的、对整个系统来说重要的战略性决定，是必然性的体现；而具体层次则含有一些次要的与实现有关的算法和逻辑，以及战术性的决定，带有相当大的偶然性选择。具体层次的代码是会经常有变动的，不能避免出现错误。抽象层次依赖于具体层次，使许多具体层次的细节的算法变化立即影响到抽象层次的宏观的商务逻辑，导致微观决定宏观，战术决定战略，偶然决定必然。如下图所示，这难道不是很荒唐的吗？



抽象层次含有宏观的和重要的商务逻辑，难道不应当由它决定具体层次的实现和具体算法的改变吗？抽象层次含有战略的决策，难道不应当由它来决定具体层次的战术的决策吗？抽象层次含有必然性的选择，难道不应当由它来指导具体层次的偶然性选择吗？

抽象层次依赖于具体层次显然是不对的。依赖倒转原则（Dependence Inversion Principle 或简称为 DIP），就是要把错误的依赖关系再倒转过来，如下图所示。



依赖倒转原则是 COM、CORBA、JavaBean 以及 EJB 等构建设计模型背后的基本原则。

8.2 复用与可维护性的"倒转"

从复用的角度来看，高层次的模块是设计者应当复用的。但是在传统的过程性的设计中，复用却侧重于具体层次模块的复用，比如算法的复用、数据结构的复用、函数库的复用等，都不可避免是具体层次模块里的复用。较高层次的结构依赖于较低层次的结构，较低层次的结构又进一步依赖于更低层次的结构，如此继续，直到依赖于每一行的代码。较低层次上的修改，会造成较高层次的修改，直到高层次逻辑的修改。

同样，传统的做法也强调具体层次上的可维护性，包括一个函数、数据结构等的可维护性，而不是高级层次上的可维护性。

从复用的意义上讲，既然抽象层次含有一个应用系统最重要的宏观商务逻辑，是做战略性判断和决定的地方，那么抽象层次就应当是较为稳定的，应当是复用的重点。由于现有的复用侧重于具体模块和细节的复用，因此，"倒转"一词则是指复用应当将复用的重点放在抽象层次上。如果抽象层次的模块相对独立于具体层次的模块的话，那么抽象层次的模块的复用便是相对较为容易的了。

同样，最重要的宏观商务逻辑也应当是维护的重点，而不是相反。

因此，遵守依赖倒转原则会带来复用和可维护性的"倒转"。好了，现在就来看一看依赖倒转原则是怎样表述的吧。

8.3 依赖倒转原则

首先来看一看依赖（或者耦合）关系有什么种类。

三种耦合关系

在面向对象的系统里，两个类之间可以发生三种不同的耦合关系：

- 零耦合（Nil Coupling）关系：如果两个类没有耦合关系，就称之为零耦合。
- 具体耦合（Concrete Coupling）关系：具体性耦合发生在两个具体的（可实例化的）类之间，经由一个类对另一个具体类的直接引用造成。
- 抽象耦合（Abstract Coupling）关系：抽象耦合关系发生在一个具体类和一个抽象类（或者 Java 接口）之间，使两个必须发生关系的类之间存有最大的灵活性。

什么是依赖倒转原则

简单地说，依赖倒转原则（Dependence Inversion Principle）要求客户端依赖于抽象耦

合。依赖倒转原则的表述是：

抽象不应当依赖于细节；细节应当依赖于抽象。（Abstractions should not depend upon details. Details should depend upon abstractions）

依赖倒转原则的另一种表述是：

要针对接口编程，不要针对实现编程。（Program to an interface, not an implementation）

第二种表述是[GOF95]一书所强调的。

针对接口编程的意思就是说，应当使用 Java 接口和抽象 Java 类进行变量的类型声明、参量的类型声明、方法的返还类型声明，以及数据类型的转换等。

不要针对实现编程的意思就是说，不应当使用具体 Java 类进行变量的类型声明、参量的类型声明、方法的返还类型声明，以及数据类型的转换等。

要保证做到这一点，一个具体 Java 类应当只实现 Java 接口和抽象 Java 类中声明过的方法，而不应当给出多余的方法。

倒转依赖关系强调一个系统内的实体之间关系的灵活性。基本上，如果设计师希望遵守"开–闭"原则，那么倒转依赖原则便是达到要求的途径。

变量的静态类型和真实类型

变量被声明时的类型叫做变量的静态类型（Static Type），有些作者把静态类型叫做明显类型（Apparent Type），变量所引用的对象的真实类型叫做变量的实际类型（Actual Type）。其源代码如代码清单 1 所示。

代码清单 1：创建 employees 类的实例

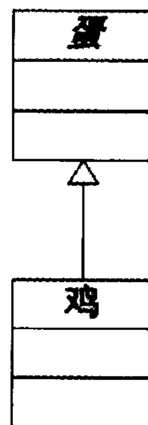
```
List employees = new Vector();
```

显然，在上面的代码中，employees 变量的静态类型是 List，而它的实际类型是 Vector。

引用对象的抽象类型

在很多情况下，一个 Java 程序需要引用一个对象。这个时候，如果这个对象有一个抽象类型的话，应当使用这个抽象类型作为变量的静态类型。这就是针对接口编程的含义。

如果"蛋"代表抽象，"鸡"代表具体，那么仍然套用"先有鸡还是先有蛋"的话题，依赖倒转原则相当于在说，"鸡"应当依赖于"蛋"，而"蛋"不应当依赖于"鸡"，先有蛋，后有鸡，如右图所示。



假设"蛋"就是 Java 接口或者 Java 抽象类，"鸡"是一个具体类，*X* 是一个变量，那么变量的声明应当是下面的样子。

```
蛋 x = new 鸡();
```

而不应当是：

```
鸡 x = new 鸡();
```

使用读者可能更加常见的创建 employee 类的例子说明的话，就是尽量不要使用下面的声明语句：

```
Vector employees = new Vector();
```

而应当使用下面的声明语句：

```
List employees = new Vector();
```

这两者的区别就是前者使用一个具体类作为变量的类型，而后者使用一个抽象类型（List 是一个 Java 接口）作为类型。后者的好处，就是在决定将 Vector 类型转换成 ArrayList 时，需要改动得很少：

```
List employees = new ArrayList();
```

这样一来，程序具有更好的灵活性，因为除去调用构造子的一行语句之外，程序的其余部分根本察觉不到有什么变化（假设程序不需要多线程的同步化）。

只要一个被引用的对象存在抽象类型，就应当在任何引用此对象的地方使用抽象类型，包括参量的类型声明、方法返还类型的声明、属性变量的类型声明等。

对象的创建

前面的两个例子同时还演示了对象创建的过程。显然，虽然 employees 的静态类型是抽象的，但是实例创建过程所调用的构造子仍然必须是具体类型的构造子。

一般而言，在创建一个对象时，Java 语言要求使用 new 关键词以及这个类本身。而一旦这个对象已经被创建出来，那么就可以灵活地使用这个对象的抽象类型来引用它。因此，Java 语言中创建一个对象的过程是违背“开–闭”原则以及依赖倒转原则的。虽然在这个类被创建出来以后，可以通过多态性使得客户端依赖于其抽象类型。

正是由于这个原因，设计模式给出了多个创建模式，特别是几个工厂模式，用于解决对象创建过程中的依赖倒转问题。

8.4　怎样做到依赖倒转原则

以抽象方式耦合是依赖倒转原则的关键。由于一个抽象耦合关系总要涉及具体类从抽象类继承，并且需要保证在任何引用到基类的地方都可以改换成其子类，因此，里氏代换原则是依赖倒转原则的基础。

在抽象层次上的耦合虽然有灵活性，但也带来了额外的复杂性。在某些情况下，如果一个具体类发生变化的可能性非常小，那么抽象耦合能发挥的好处便十分有限，这时使用具体耦合反而会更好。

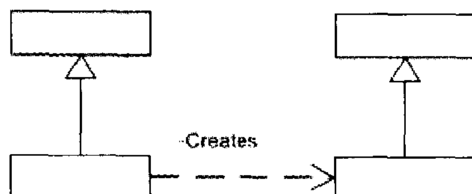
依赖倒转原则是 OO 设计的核心原则，设计模式的研究和应用是以依赖倒转原则为指导原则的。下面就举几个设计模式的例子加以说明。



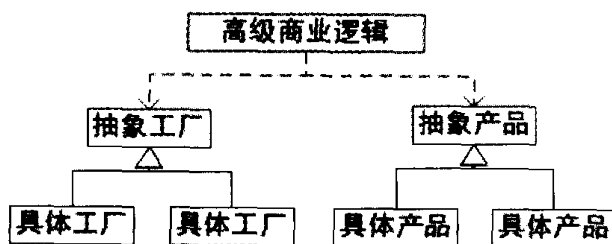
工厂方法模式

正如前面所谈到的，应当使消费一个对象的客户端只依赖于对象的抽象类型，而不是它的具体类型。但是，Java 语言要求在将一个（具体）类实例化的时候，必须调用这个具体类的构造子，所以 Java 语言给出的类的实例化方法无法做到只依赖于抽象类型。

但是设计模式给出了解决这个问题的可行方案，其中最重要的方案就是工厂模式。工厂方法模式是几个工厂模式中最为典型的一个，下图所示就是工厂方法模式的简略类图。

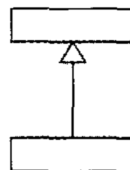


工厂模式将创建一个类的实例的过程封装起来，消费这个实例的客户端仅仅得到实例化的结果，以及这个实例的抽象类型。当然，任何方法都无法回避 Java 语言所要求的 new 关键字和直接调用具体类的构造子的做法。简单工厂模式将这个违反“开–闭”原则以及依赖倒转原则的做法封装到了一个类里面，而工厂方法模式将这个违反原则的做法推迟到了具体工厂角色中，如下图所示。



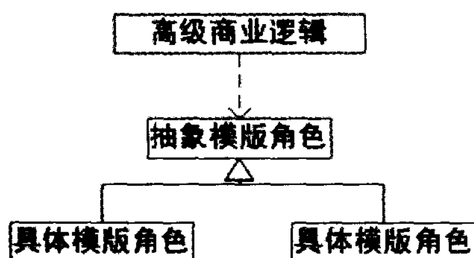
这样，通过适当的封装，工厂模式可以净化大部分的结构，而将违反原则的做法孤立到易于控制的地方。

模版方法模式



模版方法模式是依赖倒转原则的具体体现，模版方法模式的简略类图如右图所示。

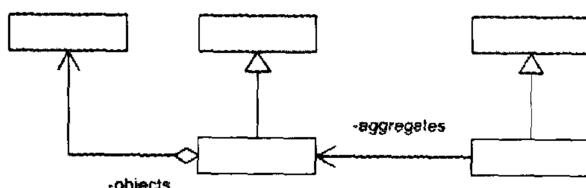
在模版方法模式里面，有一个抽象类将重要的宏观逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的具体细节上的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。模版方法模式支持依赖倒转原则，如下图所示。



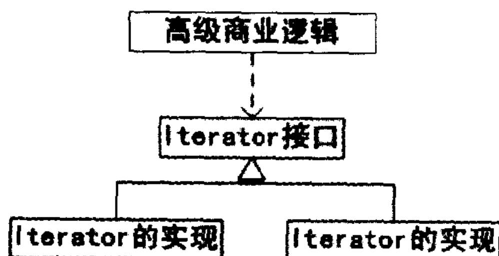
具体子类不能影响抽象类的宏观逻辑，而抽象逻辑的改变则会导致细节逻辑的改变。

迭代子模式

迭代子模式用一个工厂方法向客户端提供一个聚集的内部迭代功能，客户端得到的是一个 Iterator 抽象类型，并不知道迭代子的具体实现以及聚集对象的内部结构。迭代子模式的简略类图如下图所示。



这样一来，聚集的内部结构的改变就不会波及到客户端，从而实现了对抽象接口的依赖，如下图所示。



8.5 Java 对抽象类型的支持

在 Java 语言中，可以定义一种抽象类型，并且提供这一抽象类型的各种具体实现。实际上，Java 语言提供了两种而不是一种机制做到这一点。它们就是 Java 接口和 Java 抽象类。

Java 接口与 Java 抽象类的区别如下：

（1）这两者最明显的区别，就在于 Java 抽象类可以提供某些方法的部分实现，而 Java 接口则不可以。这大概是 Java 抽象类惟一的优点。

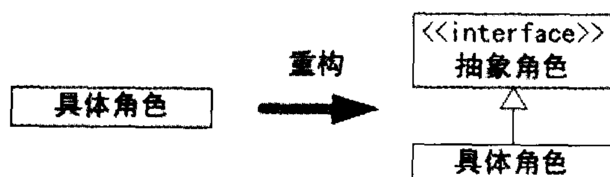


如果向一个抽象类加入一个新的具体方法，那么所有的子类型一下子就都得到了这个新的具体方法，而 Java 接口做不到这一点。如果向一个 Java 接口加入一个新的方法的话，所有实现这个接口的类就不能全部成功地通过编译了，因为它们都没有实现这个新声明的方法。这显然是 Java 接口的一个缺点。

（2）一个抽象类的实现只能由这个抽象类的子类给出，也就是说，这个实现处在抽象类所定义出的继承的等级结构中，而由于 Java 语言限制一个类只能从最多一个超类继承，因此将抽象类作为类型定义工具的效能大打折扣。

反过来看 Java 接口的情况，就会发现任何一个实现了一个 Java 接口所规定的方法的类都可以具有这个接口的类型，而一个类可以实现任意多个 Java 接口。这便是这两者之间最为重要的区别。此外，Java 接口还具有其他的优越性[BLOCH01]。

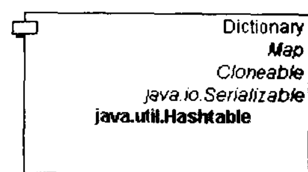
（3）从代码重构的角度上讲，将一个单独的 Java 具体类重构成一个 Java 接口的实现是很容易的。如下图所示，只需要声明一个 Java 接口，并将重要的方法添加到接口声明中，然后在具体类定义语句后面加上一个合适的 implements 子句就可以了。



而为一个已有的具体类添加一个 Java 抽象类作为抽象类型却不那么容易，因为这个具体类有可能已经有一个超类。这样一来，这个新定义的抽象类只好继续向上移动，变成这个超类的超类，如此循环，最后这个新定义的抽象类必定处于整个类型等级结构的最上端，从而使等级结构中的所有成员都会受到影响。

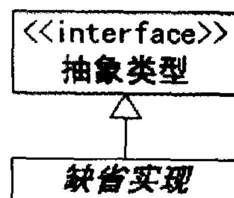
（4）Java 接口是定义混合类型（Mixin Type）的理想工具。所谓混合类型，就是在一个类的主类型之外的次要类型。一个混合类型表明一个类不仅仅具有某个主类型的行为，而且具有其他的次要行为。

比如 Hashtable 类就具有多个类型。它的主要类型是 Map，这是一种 Java 聚集。而 Cloneable 接口则给出一个次要类型，这个类型说明这个类的实例是可以安全地克隆的。同样，Serializable 也是一个次要类型，它表明这个类的实例是可以串行化的，如右图所示。



联合使用 Java 接口和 Java 抽象类

由于 Java 抽象类具有提供缺省实现的优点，而 Java 接口具有其他所有的优点，如右图所示，所以联合使用两者就是一个很好的选择。



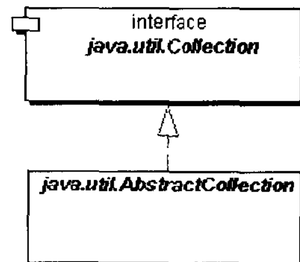
首先，声明类型的工作仍然是由 Java 接口承担的，但是同时给出的还有一个 Java 抽象类，为这个接口给出一个缺省实现。其他同属于这个抽象类型的具体类可以选择实现这个 Java 接口，也可以选择继承自这个抽象类。

如果一个具体类直接实现这个 Java 接口的话，它就必须自行实现所有的接口；相反，如果它继承自抽象类的话，它可以省去一些不必要的方法，因为它可以从抽象类中自动得到这些方法的缺省实现。

如果需要向 Java 接口加入一个新的方法的话，那么只要同时向这个抽象类加入这个方法的一个具体实现就可以了，因为所有继承自这个抽象类的子类都会从这个抽象类得到这个具体方法。

这其实就是缺省适配模式，关于这个模式，读者可以参阅本书的"缺省适配（Default Adapter）模式"一章。

在 Java 语言 API 中也是用了这种缺省适配模式，而且全都遵循一定的命名规范：Abstract + 接口名。比如对于接口 Collection，抽象类的名字是 AbstractCollection，如右图所示。

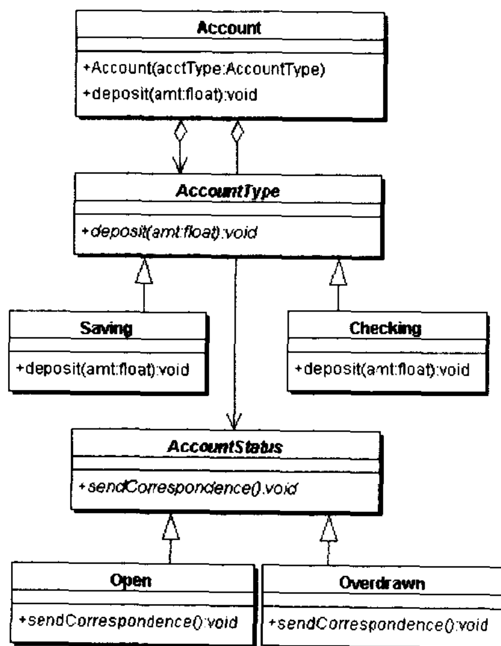


其他的例子还有 Map 与 AbstractMap，List 与 AbstractList，以及 Set 与 AbstractSet 等。这种联合使用接口和抽象类的做法可以充分利用两者的优点，克服两者的缺点。

8.6 一个例子：账号、账号的种类和账号的状态

在下面的例子里，Account 类有两个聚合关系，一个指向 AccountType，另一个指向 AccountStatus。然而，这两个类型均是抽象类型。每一个抽象类型均有多于一个的具体实现。比如 AccountType 有 Saving 和 Checking 两种具体子类；而 AccountStatus 则有 Open 和 Overdrawn 两种具体实现。

在本例子里面，Account 持有静态类型为 AccountType 和 AccountStatus 的两个对象的引用，如下图所示。





Account 类的源代码如代码清单 2 所示。

代码清单 2：Account 类的源代码

```
public class Account
{
    private AccountType accountType;
    private AccountStatus accountStatus;
    public Account(AccountType acctType)
    {
        //write your code here
    }
    public void deposit(float amt)
    {
        //write your code here
    }
}
```

抽象类 AccountType 定义出所有具体子类必须实现的接口，如代码清单 3 所示。

代码清单 3：抽象类 AccountType 的源代码

```
abstract public class AccountType
{
    public abstract void deposit(float amt);
}
```

抽象类 AccountStatus 定义出所有的具体子类必须实现的接口，如代码清单 4 所示。

代码清单 4：抽象类 AccountStatus 的源代码

```
abstract public class AccountStatus
{
    public abstract void sendCorrespondence();
}
```

下面就是一个具体的 AccountType 的子类，它实现了抽象超类 AccountType 所声明的接口，如代码清单 5 所示。

代码清单 5：具体实现类 Saving 的源代码

```
public class Savings extends AccountType
{
    public void deposit(float amt)
    {
        //write your code here
    }
}
```

Saving 类代表储蓄账号，是 AccountType 的子类。

而 Checking 是 AccountType 的另一个具体子类，它也实现了抽象超类 AccountType 所

声明的接口如代码清单 6 所示。

代码清单 6：具体实现类 Checking 的源代码

```
public class Checking extends AccountType
{
    public void deposit(float amt)
    {
        //write your code here
    }
}
```

Checking 类代表支票账号，是 AccountType 的子类。

下面给出的是抽象类 AccountStatus 的一个具体子类，表示账号处于"开"状态的状态类 Open，如代码清单 7 所示。

代码清单 7：具体实现类 Open 的源代码

```
public class Open extends AccountStatus
{
    public void sendCorrespondence()
    {
        //write your code here
    }
}
```

下面是抽象类 AccountStatus 的另一个具体子类，表示账号处于"超支"状态的状态类 Overdrawn，如代码清单 8 所示。

代码清单 8：具体实现类 Overdrawn 的源代码

```
public class Overdrawn extends AccountStatus
{
    public void sendCorrespondence()
    {
        //write your code here
    }
}
```

为了创建一个实例，必须直接调用此具体类的构造子。如果需要将一个具体类替换成为另一个具体类，而不改变创建此实例的方法，只有一个办法，那就是将创建责任下推给一个工厂类。

在这个例子里，Account 类依赖于 AccountType 和 AccountStatus 两个抽象类型，而不是它们的子类型。AccountType 有两个子类型：

- 储蓄账号：以 Savings 具体类代表。
- 支票账号：以 Checking 具体类代表。

AccountStatus 也有两个子类型：

- 开状态：以 Open 具体类代表。



● 超支状态：以 Overdrawn 具体类代表。

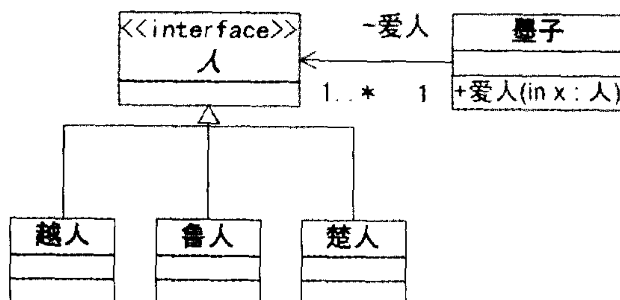
Account 类并不依赖于具体类，因此当有新的具体类型添加到系统中时，Account 类不必改变。例如，如果系统引进了一种新型的账号：MoneyMarket 类型，Account 类以及系统里面所有其他的依赖于 AccountType 抽象类的客户端类均不必改变。MoneyMarket 的源代码如代码清单 9 所示。

代码清单 9：具体类 MoneyMarket 的源代码

```
public class MoneyMarket extends AccountType
{
    public void deposit(float amt)
    {
        //write your code here
    }
}
```

8.7　墨子论“取周”

《墨子·小取》中说，“爱人，待周爱人而后为爱人”。换言之，如果“爱人”是一个行为的话，这个行为对所有的人成立。因此，“爱人”之人是抽象之人，所有具体之人皆是抽象“人”的子类型，所以爱人的行为对所有的子类型成立，如下图所示。



因此，“取周”便是抽象依赖之意，抽象的依赖使得“爱人”行为对所有的人，包括见过的和没有见过的人都成立。

8.8　依赖倒转原则的优缺点

依赖倒转原则虽然很强大，但却是最不容易实现的。因为依赖关系倒转的缘故，对象的创建很可能要使用对象工厂，以避免对具体类的直接引用，此原则的使用还会导致大量的类。对不熟悉面向对象技术的工程师来说，维护这样的系统需要较好地面向对象的设计的知识。

此外，依赖倒转原则假定所有的具体类都是会变化的，这也不总是正确的。有一些具体类可能是相当稳定、不会发生变化的，消费这个具体类实例的客户端完全可以依赖于这

个具体类型，而不必为此发明一个抽象类型。

参考文献

[MARTIN95] Martin C. Robert. Object Oriented Design Quality Metrics: An Analysis of dependencies. ROAD, Vol. 2, No. 3, Sep-Oct, 1995

[MARTIN00] Martin C. Robert . Design Principles and Design Patterns. http://www.objectmentor.com, 2000

[MOZI00] 孙波（注释）. 墨子全文注释本. 华夏出版社，2000

[KIRK02] Kirk Knoernschild. Java Design – Objects, UML, and Process. Addison-Wesley, 2002

[BLOCH01] Joshua Bloch. Effective Java – Programming Language Guide. published by Addison-Wesley, 2001

第 9 章 接口隔离原则（ISP）

接口隔离原则（Interface Segregation Principle，常常略写做 ISP）讲的是：使用多个专门的接口比使用单一的总接口要好。

换言之，从一个客户类的角度来讲：一个类对另外一个类的依赖性应当是建立在最小的接口上的。

9.1 什么是接口隔离原则

正如本书在“专题：Java 接口”一章中所指出的那样，人们所说的“接口”往往是指两种不同的东西：一种是指 Java 语言中的有严格定义的 Interface 结构，比如 java.lang.Runnable 就是一个 Java 接口；另一种就是一个类型所具有的方法特征的集合，也称做“接口”，但仅是一种逻辑上的抽象。

对应于这两种不同的用词，接口隔离原则的表达方式以及含义都有所不同。

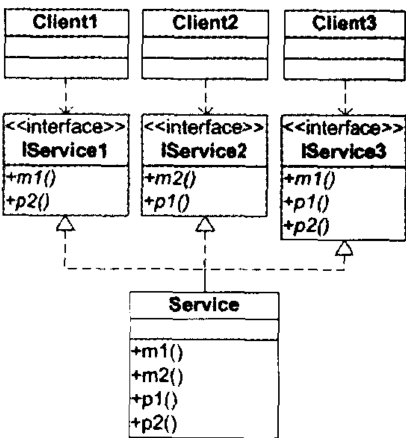
角色的合理划分

将“接口”理解为一个类所提供的所有方法的特征集合，也就是一种在逻辑上才存在的概念。这样的话，接口的划分就直接带来类型的划分。

一个接口相当于剧本中的一种角色，而此角色在一个舞台上由哪一个演员来演则相当于接口的实现。因此，一个接口应当简单地代表一个角色，而不是多个角色。如果系统涉及到多个角色的话，那么每一个角色都应当由一个特定的接口代表。

为避免混淆，本书将这种角色划分的原则叫做角色隔离原则。

定制服务



将接口理解成为狭义的 Java 接口，这样一来，接口隔离原则讲的就是为同一个角色提供宽、窄不同的接口，以对付不同的客户端，如右图所示。这种办法在服务行业中叫做定制服务（Customized Service），这也是本书给这种诠释的一个名字。

在上面的示意性类图中，有一个角色 Service 以及三个不同的客户端。这三个客户端需要的服务都是稍稍不同的，因此系统分别为它们提供了三个不同的 Java 接口，即 IService1，IService2 以及

IService3。显然，每一个 Java 接口都仅仅将客户端需要的行为暴露给客户端，而没有将客户端所不需要的行为放到接口中。

熟悉适配器模式的读者可以辨认出，这是适配器模式的应用。

接口污染

过于臃肿的接口是对接口的污染（Interface Contamination）。

由于每一个接口都代表一个角色，实现一个接口的对象，在它的整个生命周期中都扮演这个角色，因此将角色区分清楚就是系统设计的一个重要工作。因此，一个符合逻辑的推断，不应当将几个不同的角色都交给同一个接口，而应当交给不同的接口。

一个没有经验的设计师往往想节省接口的数目，因此，将一些看上去差不多的接口合并。一些人将这看做是代码优化的一部分，这是错误的。

准确而恰当地划分角色以及角色所对应的接口，是面向对象的设计的一个重要的组成部分。将没有关系的接口和并在一起，形成一个臃肿的大接口，是对角色和接口的污染。

与迪米特法则的关系

迪米特法则要求任何一个软件实体，除非绝对需要，不然不要与外界通信。即使必须进行通信，也应当尽量限制通信的广度和深度。

显然，定制服务原则拒绝向客户端提供不需要提供的行为，是符合迪米特法则的。

9.2 一个角色隔离原则的例子

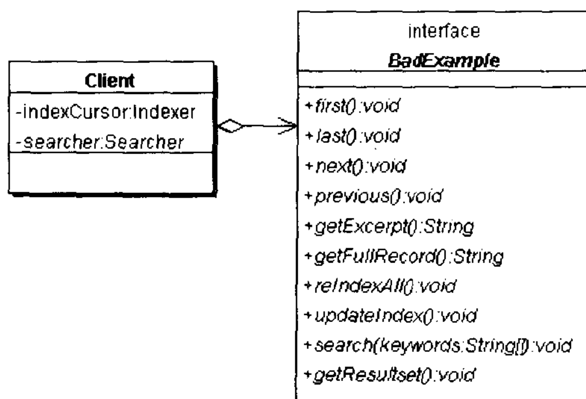
本节从代码重构的角度讨论怎样将一个臃肿的角色重新分割成更为合适的较小角色。

全文查询引擎的系统设计

本章在这里以一个网站的全文查询引擎的系统设计为例，这个例子取材自一个真实的项目，如果读者使用过 AltaVista 公司提供的文字搜索引擎软件包的话，就会很熟悉本例子的情形。一个动态的资料网站将大量的文件资料存储在文件中或关系数据库里面，用户可以通过输入一个和数个关键词进行全网站的全文搜索。这个搜索引擎需要维持一个索引库，在本例子里面索引库以文本文件方式存于文件系统中。在源数据被修改、删除或增加时，搜索引擎要做相应的动作，以保证引擎的索引文件也被相应地更新。

反面例子

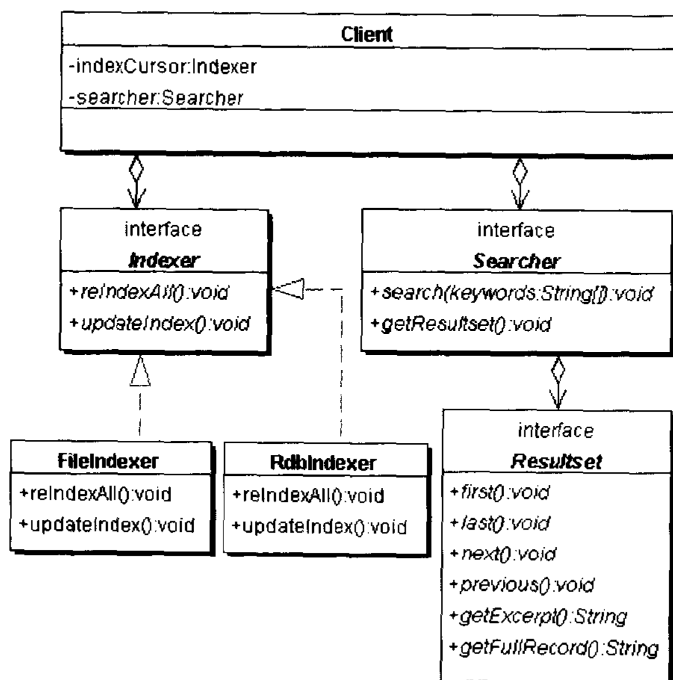
首先，下图所示为一个不好的解决方案。一个叫做 BadExample 的接口负责所有的操作，从提供搜索功能到建立索引的功能，甚至包括搜索结果集合的功能均在一个接口内提供。



这个解决方案违反了角色分割原则，把不同功能的接口放在一起，由一个接口给出包括搜索器角色、索引生成器角色以及搜索结果集角色在内的所有角色。

角色的分割

那么，遵守接口隔离原则的做法是怎样的呢？如下图所示。



在图中可以看出，搜索引擎的功能被分割为三个角色：

- 搜索器角色
- 索引生成器角色
- 搜索结果集角色

以索引生成器角色为例，由于索引生成因数据的格式不同而不同，故分为 RdbIndexer 和 FileIndexer 两种实现。FileIndexer 类代表对诸如*.txt、*.html、*.doc 以及*.pdf 等文件类型的数据生成全文索引，而 RdbIndexer 则针对关系数据库的数据进行全文索引生成。这两个实现扮演的同为索引生成器角色，就好像扮演同样角色的两个不同演员一样。

搜索器角色则是与索引生成器角色完全不同的角色，它提供用户全文搜索功能。用户传进一些关键字，搜索器角色则返回一个 ResultSet 对象。

搜索结果集角色就是 ResultSet。它给用户提供对集合进行迭代走访的功能，如 first()将光标移到集合的第一个元素；last()将光标移到集合的最后一个元素；next()将光标移到集合的下一个元素；previous()将光标移到集合的前一个元素；而 getExcerpt()则返回当前记录的摘要；而 getFullRecord()则将记录的全文返回。

9.3 定制服务的例子

定制服务（Customized Service）也是一个重要的设计原则。它的意思是说，如果客户端仅仅需要某一些方法的话，那么就应当向客户端提供这些需要的方法，而不要提供不需要的方法。

这样做的效果是什么呢？

（1）这样做很整洁。从美学的角度上考虑，这是一个很好的做法。从这样的一个设计可以看出，设计师花了很多的时间在分析和划分这些接口上面。

但是这并不是最重要的效果，没有人会仅仅因为美学效果而将这一原则当做面向对象的设计原则。

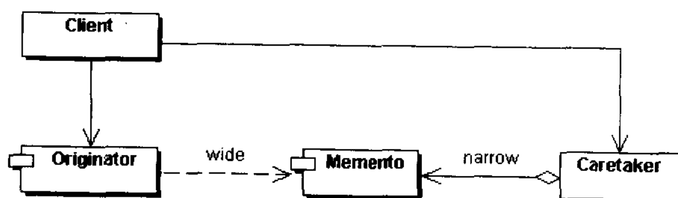
（2）系统的可维护性。向客户端提供的 public 接口是一种承诺，一个 public 接口一旦提供，就很难撤回。作为软件提供商，没有人愿意做出过多的承诺，特别是不必要的承诺。过多的承诺会给系统的维护造成不必要的负担。

如果这些接口仅仅是提供给公司内部的系统使用，那么将这些接口隔离开来，也可以降低维护的成本。因为如果一旦所提供的服务出现变化的话，设计师知道哪些客户端会受到影响，哪些不会受到影响。

这显然也是符合迪米特法则的。

备忘录模式

备忘录模式（Memento Pattern）的用意是在不破坏封装的条件下，捕捉一个对象的状态，并将之外部化，从而可以在将来合适的时候把这个对象还原到存储起来的状态。备忘录模式的简略类图如下图所示。



在这里，不破坏封装是一个关键词。为了做到这一点，必须使备忘录对象向外界提供双重接口，也即一个窄接口和一个宽接口。

宽接口是为发起人角色准备的，因为这个备忘录角色所存储的状态就是属于这个发起人角色的，而且这个角色需要访问备忘录角色所存储的信息以便恢复自己的状态。

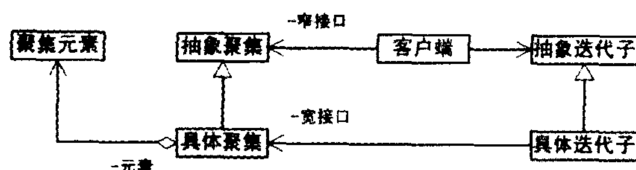
窄接口是为包括负责人角色在内的所有其他对象准备的，因为它们不需要、也不应该读取备忘录角色所存储的信息。

换言之，发起人角色和负责人角色就相当于备忘录角色的不同客户端，而这种为不同客户端提供不同接口的做法就是定制服务概念的体现。

关于备忘录模式，请读者参见本书的“备忘录（Memento）模式”一章。

迭代子模式

迭代子模式提供一个迭代子对象，使得客户端可以顺序地访问一个聚集中的元素，而不必暴露聚集的内部表象。迭代子模式的示意图如下图所示。



换言之，上面的这个系统的客户端和系统内部的迭代子对象都需要访问聚集对象，但是它们所需要的访问性质有所不同。前者仅需要通过一个迭代子接口遍历聚集的元素，而迭代子对象则需要知道聚集对象的内部结构信息。

因此，聚集对象向不同的客户端提供了不同的接口，一个是宽接口，提供给迭代子对象；另一个是窄接口，提供给系统的客户端。

关于迭代子模式，请读者参见本书的“迭代子（Iterator）模式”一章。

问答题

请从接口隔离原则出发分析一下“看人下菜碟”这个说法的意义。

问答题答案

“看人下菜碟”的意思是说，要看客人是谁，再提供不同档次的饭菜。

从接口隔离原则的角度出发，根据客户需要的不同，而为不同的客户端提供不同的服务是一种应当得到鼓励的做法。

参考文献

[MEYER88] Bertrand Meyer. Object Oriented Software Construction. Prentice Hall, 1988

[MARTIN96] Robert C. Martin. Engineering Notebook——C++ Report. Nov-Dec, 1996

[KIRK02] Kirk Knoernschild. Java Design - Objects, UML, and Process. Addison-Wesley, 2002

第 10 章　合成/聚合复用原则（CARP）

合成/聚合复用原则（Composite/Aggregate Reuse Principle，或 CARP）经常又叫做合成复用原则（Composite Reuse Principle 或 CRP）。合成/聚合复用原则就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；新的对象通过向这些对象的委派达到复用已有功能的目的。

这个设计原则有另一个更简短的表述：要尽量使用合成/聚合，尽量不要使用继承。

合成（Composite）一词的使用很广泛，经常导致混淆。为避免这些混淆，不妨先来考察一下“合成”与“聚合”的区别。

10.1　合成和聚合的区别

合成（Composition）和聚合（Aggregation）均是关联（Association）的特殊种类。聚合用来表示“拥有”关系或者整体与部分的关系；而合成则用来表示一种强得多的“拥有”关系。在一个合成关系里，部分和整体的生命周期是一样的。一个合成的新的对象完全拥有对其组成部分的支配权，包括它们的创建和湮灭等。使用程序语言的术语来讲，组合而成的新对象对组成部分的内存分配、内存释放有绝对的责任。

更进一步来讲，一个合成的多重性（Multiplicity）不能超过 1，换言之，一个合成关系中的成分对象是不能与另一个合成关系共享的。一个成分对象在同一个时间内只能属于一个合成关系。如果一个合成关系湮灭了，那么所有的成分对象要么自己湮灭所有的成分对象（这种情况较为普遍），要么就得将这一责任交给别人（这种情况较为罕见）。

用 C 程序员较易理解的语言来讲，合成是值的聚合（Aggregation by Value），而通常所说的聚合则是引用的聚合（Aggregation by Reference）。

10.2　复用的基本种类

在面向对象的设计里，有两种基本的办法可以在不同的环境中复用已有的设计和实现，即通过合成/聚合或通过继承。那么这两种不同的复用方式在可维护性上面有何区别呢？

合成/聚合复用

由于合成或聚合可以将已有的对象纳入到新对象中，使之成为新对象的一部分，因此新的对象可以调用已有对象的功能。这样做有下面的好处：

- 新对象存取成分对象的惟一方法是通过成分对象的接口。
- 这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的。
- 这种复用支持包装。
- 这种复用所需的依赖较少。
- 每一个新的类可以将焦点集中在一个任务上。
- 这种复用可以在运行时间内动态进行，新对象可以动态地引用与成分对象类型相同的对象。

一般而言，如果一个角色得到了更多的责任，那么可以使用合成/聚合关系将新的责任委派到合适的对象。

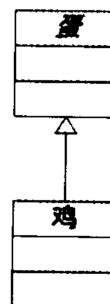
当然，这种复用也有缺点。其中最主要的缺点就是通过使用这种复用建造的系统会有较多的对象需要管理。

通过继承达到复用的目的

合成/聚合作为复用的手段可以应用到几乎任何环境中去。而与合成/聚合不同的是，继承只能应用到很有限的一些环境中去。换言之，尽管集成是一种非常重要的复用手段，但是设计师应当首先考虑使用合成/聚合，而不是继承。

继承的种类

继承是面向对象的语言特有的复用工具，而且是最容易被滥用的复用工具。这里讨论的继承，是指从一个 Java 类到另一个 Java 类的实现性继承，也就是实现继承，并不包括接口继承。一个实现继承的例子如右图所示。



继承复用通过扩展一个已有对象的实现来得到新的功能，基类明显地捕获共同的属性和方法，而子类通过增加新的属性和方法来扩展超类的实现。继承是类型的复用，比如下面都是继承的例子：

- 男人和女人是人类。
- 上推排序（Bubble Sort）是排序程序的一种。
- 汽车驾照是官方文件的一种。
- 正式雇员和临时雇员均是雇员的一种。
- 经理是正式雇员的一种。

在面向对象的设计理论早期，设计师十分热衷于继承，好像继承就是最好的复用手段。随着时间的推移和实践经验的积累，人们逐渐认识到了继承关系的缺点。

继承复用的优点

利用继承关系达到复用的做法有下面的优点：

- 新的实现较为容易，因为超类的大部分功能可以通过继承关系自动进入子类。
- 修改或扩展继承而来的实现较为容易。

继承复用的缺点

与合成/聚合复用不同的是，继承有多个缺点：

- 继承复用破坏包装，因为继承将超类的实现细节暴露给子类。由于超类的内部细



节常常是对子类透明的，因此这种复用是透明的复用，又称“白箱”复用。

- 如果超类的实现发生改变，那么子类的实现也不得不发生改变。因此，当一个基类发生改变时，这种改变会像水中投入石子引起的水波一样，将变化一圈又一圈地传导到一级又一级的子类，使设计师不得不相应地改变这些子类，以适应超类的变化。
- 从超类继承而来的实现是静态的，不可能在运行时间内发生改变，因此没有足够的灵活性。

由于以上的这些缺点，尽量使用合成/聚合而不是继承来达到对实现的复用，是非常重要的设计原则。

10.3　从代码重构的角度理解

在很多情况下，缺乏经验的 Java 设计师之所以选择继承关系描述两个类之间的关系，是因为对继承关系的理解不够造成的。而要正确地使用继承关系，必须透彻地理解里氏代换原则和 Coad 法则。

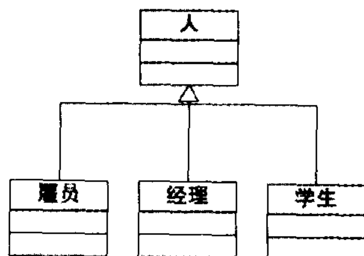
一般来说，对违反里氏代换原则的设计进行重构时，可以采取两个办法：一是加入一个抽象超类，这一办法已经在本书的“里氏代换原则”一章中讨论过了；二是将继承关系改写合成/聚合关系，这一点是本章讨论的重点。

区分“Has-A”与“Is-A”

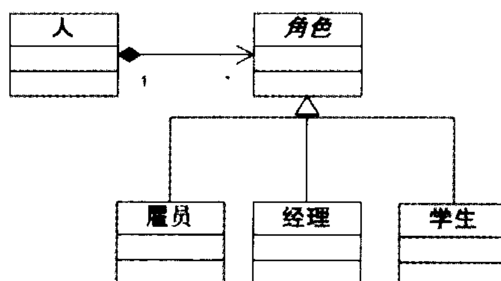
“Is-A”是严格的分类学意义上的定义，意思是一个类是另一个类的“一种”。而“Has-A”则不同，它表示某一个角色具有某一项责任。

导致错误地使用继承而不是合成/聚合的一个常见的原因是错误地把“Has-A”当做“Is-A”。“Is-A”代表一个类是另一个类的一种；“Has-A”代表一个类是另一个类的一个角色，而不是另一个类的一个特殊种类。这是 Coad 条件的第一条。

请考虑一下下图所示的类图中所描述的例子。“人”被继承到“雇员”、“经理”和“学生”等子类。而实际上，“雇员”、“经理”和“学生”分别描述一种角色，而“人”可以同时有几种不同的角色。比如，一个“人”既然是“经理”，就必然是“雇员”；而此“人”可能同时还参加 MBA 课程，从而也是一个“学生”。使用继承来实现角色，则只能使每一个“人”具有 Has-A 角色，而且继承是静态的，这会使得一个“人”在成为“雇员”身份后，就永远为“雇员”，不能称为“经理”或“学生”，而这显然是不合理的。



这一错误的设计源自于把“角色”的等级结构与“人”的等级结构混淆起来，把 Has-A 角色误解为 Is-A 角色。因此要纠正这一错误，关键是区分“人”与“角色”的区别。下图所示的设计就正确地做到了这一点。



从上图可以看出，每一个“人”都可以有一个以上的“角色”，所以一个人可以同时是“雇员”，又是“经理”，甚至同时又是“学生”。而且由于“人”与“角色”的耦合是通过合成的，因此，角色可以有动态的变化。一个“人”可以开始是一个“雇员”，然后晋升为“经理”，然后又由于他参加了 MBA 课程，又成为了“学生”。

这就是说，当一个类是另一个类的角色时，不应当使用继承描述这种关系。

与里氏代换原则联合使用

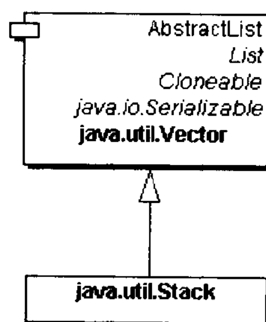
里氏代换原则是继承复用的基石。如果在任何使用 B 类型的地方都可以使用 S 类型，那么 S 类型才能称为 B 类型的子类型（Subtype），而 B 类型才能称为 S 类型的基类型（Base Type）。

换言之，只有当每一个 S 在任何情况下都是一种 B 的时候，才可以将 S 设计成为 B 的子类。如果两个类的关系是“Has-A”关系而不是“Is-A”关系，这两个类一定违反里氏代换原则。

只有两个类满足里氏代换原则，才有可能是“Is-A”关系。

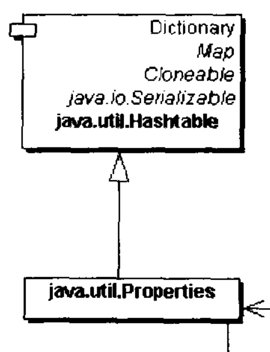
Java 语言 API 中的例子

在 Java 语言的 API 中，有几个明显违反这一原则的例子，其中最为著名的就是 Stack 和 Properties。前者被不当地设置为 Vector 的子类，如下图所示。





而 Properties 被不恰当地设置成 Hashtable 的子类，如下图所示。



一个 Stack 不是一个 Vector，所以 Stack 不应当设置成为 Vector 的子类。同样地，一个性质列（Properties）也不是一个 Hashtable。在两种情况下，使用聚合比使用继承关系更合适。

由于 Properties 继承了 Hashtable 的行为，因而当 p 是一个 Properties 类型的对象时，p.getProperties(key)与 p.get(key)就会给出不同的结果。前者来自于 Properties 本身，因此会利用默认值；而后者则来自于 Hashtable，因此不会利用默认值。

更糟糕的是，由于 Properties 是 Hashtable 的子类，因此，客户端可以通过类型的转换，直接使用超类型的行为。比如，Properties 假定所有的键和值都是 String 类型的，如果不是，就会导致运行崩溃。但是，客户端完全可以通过 Hashtable 提供的行为加入任意类型的键和值。绕过 Properties 的接口，并导致 Properties 的内部矛盾和崩溃[BLOCH01]。

这样一来，Properties 其实仅仅是有一些 Hashtable 的属性的，换言之，这是一个“Has-A”的关系，而不是一个“Is-A”的关系。

参考文献

[Lieberherr89] Karl. J. Lieberherr and I. Holland. Assuring good style for object-oriented programs. IEEE Software，September 1989

[Lieberherr96] Karl Lieberherr. Adaptive Object Oriented Software–The Demeter Method. PWS Publishing Co，Boston，1996

[COAD91] P. Coad. E. Yourdon Object-oriented Analysis. Prentice–Hall Inc，USA，1991

[KIRK02] Kirk Knoernschild. Java Design–Objects，UML，and Process. Addison-Wesley，2002

[BLOCH01] Joshua Bloch. Effective Java–Programming Language Guide. published by Addison-Wesley，2001

第 11 章　迪米特法则（LoD）

迪米特法则（Law of Demeter 或简写为 LoD）又叫做最少知识原则（Least Knowledge Principle 或简写为 LKP），就是说，一个对象应当对其他对象有尽可能少的了解。

迪米特法则最初是用来作为面向对象的系统设计风格的一种法则，于 1987 年秋天由 Ian Holland 在美国东北大学（Northeastern University）为一个叫做迪米特（Demeter）的项目设计提出的，因此叫做迪米特法则[LIEB89] [LIEB86] 。这条法则实际上是很多著名系统，比如火星登录软件系统、木星的欧罗巴卫星轨道飞船的软件系统的指导设计原则。

11.1　迪米特法则的各种表述

没有任何一个其他的 OO 设计原则像迪米特法则这样有如此之多的表述方式，下面给出的也只是众多的表述中较有代表性的几种：

- 只与你直接的朋友们通信（Only talk to your immediate friends）。
- 不要跟“陌生人”说话（Don't talk to strangers）。
- 每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。

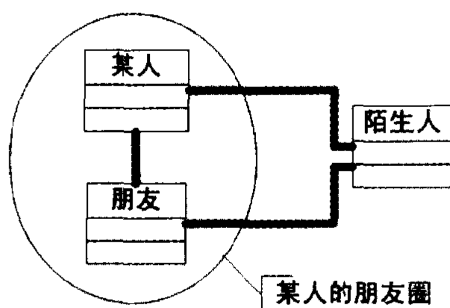
在上面的表述里面，什么是“直接”、“陌生”和“密切”则被有意地模糊化了，以便在不同的环境下可以有不同的解释。

11.2　狭义的迪米特法则

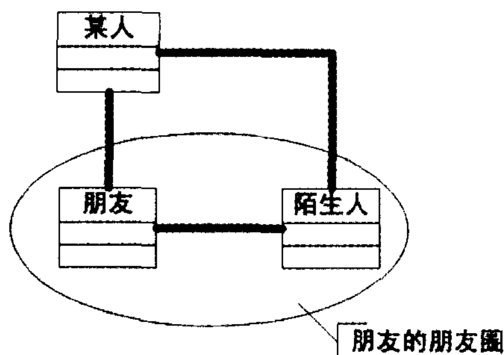
如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中的一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。

朋友圈与陌生人

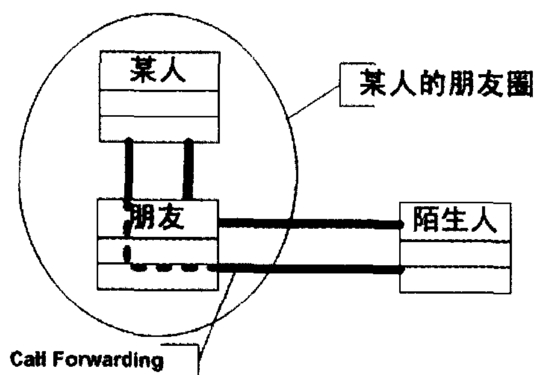
如下图所示，“某人”与一个“朋友”组成自己的朋友圈，两个人都需要与一个圈外的“陌生人”发生相互作用。



“朋友”与“陌生人”若是朋友，组成“朋友”的朋友圈如下图所示。



相比较之下，“某人”其实并不需要与“陌生人”直接发生相互作用，但是“朋友”则更需要与“陌生人”发生相互作用。这时候，迪米特法则建议“某人”不要直接与“陌生人”发生相互作用，而是通过“朋友”与之发生直接的相互作用，如下图所示。



这时候，“朋友”实际上起到了将“某人”对“陌生人”的调用转发给“陌生人”的作用。这种传递叫做调用转发（Call Forwarding）。

所谓调用转发，需要隐藏“陌生人”的存在，使得“某人”仅知道“朋友”，而不知道“陌生人”；换言之，“某人”会认为他所调用的这个方法是“朋友”的方法。

那么“某人”的朋友圈是怎样确定的呢？

朋友圈的确定

以下的条件称为“朋友”条件：

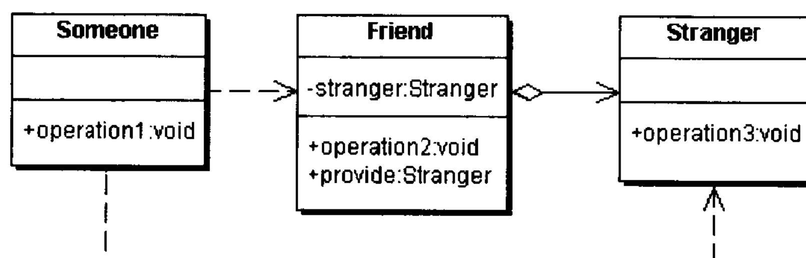
- 当前对象本身（this）
- 以参量形式传入到当前对象方法中的对象
- 当前对象的实例变量直接引用的对象
- 当前对象的实例变量如果是一个聚集，那么聚集中的元素也都是朋友
- 当前对象所创建的对象

任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”；否则就是“陌生人”[GRAND98-2]。

下面就从代码重构的角度讨论怎样将一个不符合迪米特法则的系统，重构成为符合迪米特法则的系统。

不满足迪米特法则的系统

这里要讨论的系统由三个类组成，分别是 Someone，Friend 和 Stranger。其中 Someone 与 Friend 是朋友，而 Friend 与 Stranger 是朋友。系统的结构图如下图所示。



从上面的类图可以看出，Friend 持有一个 Stranger 对象的引用，这就解释了为什么 Friend 与 Stranger 是朋友。为了解释为什么 Someone 与 Friend 是朋友，请参见这里给出 Someone 的源代码，如代码清单 1 所示。

代码清单 1：Someone 类的源代码

```
public class Someone
{
    public void operation1(Friend friend)
    {
        Stranger stranger = friend.provide();
        stranger.operation3();
    }
}
```

可以看出，Someone 具有一个方法 operation1()，这个方法接受 Friend 为参量。显然，根据“朋友”的定义，Friend 是 Someone 的朋友。其中 Friend 的 provide()方法会提供自己所创建的 Stranger 的实例，如代码清单 2 所示。

代码清单 2：Friend 类的源代码

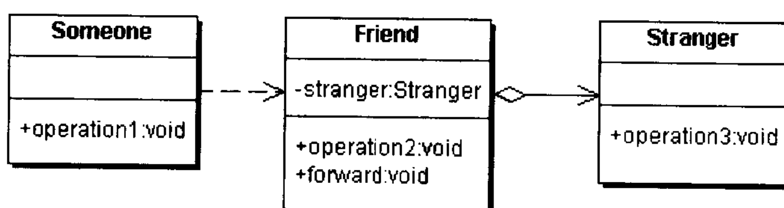
```
public class Friend
{
```

```
        private Stranger stranger = new Stranger();
        public void operation2()
        {
        }
        public Stranger provide()
        {
            return stranger;
        }
    }
```

显然，Someone 的方法 operation1()不满足迪米特法则。为什么呢？因为这个方法引用了 Stranger 对象，而 Stranger 对象不是 Someone 的朋友。

使用迪米特法则进行改造

可以使用迪米特法则对上面的例子进行改造，改造的做法就是调用转发。改造后的情况如下图所示。



从上面的类图可以看出，与改造前相比，在 Someone 与 Stranger 之间的联系已经没有了。Someone 不需要知道 Stranger 的存在就可以做同样的事情。Someone 的源代码如代码清单 3 所示。

代码清单 3：Someone 类的源代码

```
public class Someone
{
    public void operation1(Friend friend)
    {
        friend.forward();
    }
}
```

从源代码可以看出，Someone 通过调用自己的朋友 Friend 对象的 forward()方法做到了原来需要调用 Stranger 对象才能够做到的事情。那么这个 forward()方法是做什么的呢？如代码清单 4 所示。

代码清单 4：Friend 类的源代码

```
public class Friend
{
    private Stranger stranger = new Stranger();
```

```
        public void operation2()
        {
            System.out.println("In Friend.operation2()");
        }
        public void forward()
        {
            stranger.operation3();
        }
    }
```

原来 Friend 类的 forward()方法所做的就是以前 Someone 要做的事情，使用了 Stranger 的 operation3()方法，而这种 forward()方法叫做转发方法。

由于使用了调用转发，使得调用的具体细节被隐藏在 Friend 内部，从而使 Someone 与 Stranger 之间的直接联系被省略掉了。这样一来，使得系统内部的耦合度降低。在系统的某一个类需要修改时，仅仅会直接影响到这个类的“朋友”们，而不会直接影响到其余部分。

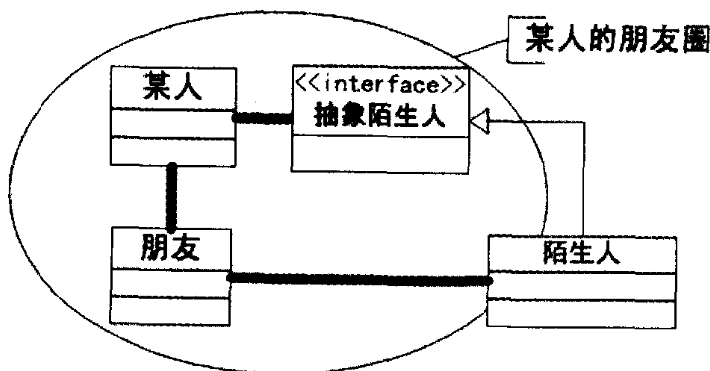
狭义的迪米特法则的缺点

遵循狭义的迪米特法则会产生一个明显的缺点：会在系统里造出大量的小方法，散落在系统的各个角落。这些方法仅仅是传递间接的调用，因此与系统的商务逻辑无关。当设计师试图从一张类图看出总体的架构时，这些小的方法会造成迷惑和困扰。

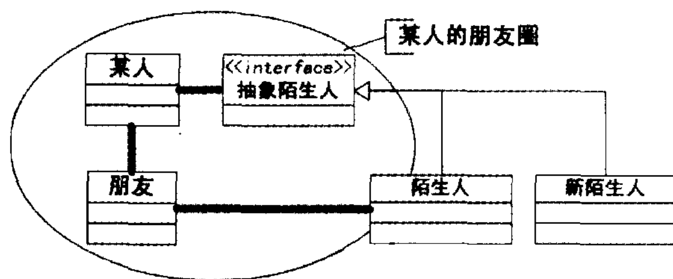
遵循类之间的迪米特法则会使一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联。但是，这也会造成系统的不同模块之间的通信效率降低，也会使系统的不同模块之间不容易协调。

与依赖倒转原则互补使用

[KIRK02]指出，为了克服狭义的迪米特法则的缺点，可以使用依赖倒转原则，引入一个抽象的类型引用“抽象陌生人”对象，使“某人”依赖于“抽象陌生人”。换言之，就是将“抽象陌生人”变成朋友，如下图所示。



"某人"现在与一个抽象角色建立了朋友关系，这样做的好处是"朋友"可以随时将具体"陌生人"换掉。只要新的具体"陌生人"具有相同的抽象类型，那么"某人"就无法区分它们。这就允许"陌生人"的具体实现可以独立于"某人"而变化，如下图所示。



在本章后面所附的问答题中给出了一个带有 Java 源代码的示意性系统。关于"依赖倒转原则"，请读者参阅本书的"依赖倒转原则（DIP）"一章。

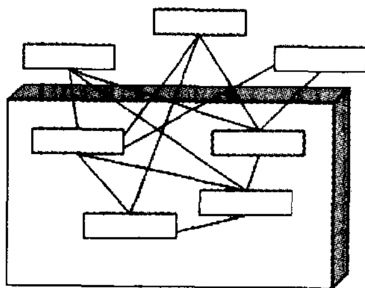
11.3 迪米特法则与设计模式

狭义的迪米特法则要求一个对象仅仅与其朋友发生相互作用。读者可以看出，门面模式和调停者模式实际上就是迪米特法则的应用。

门面模式

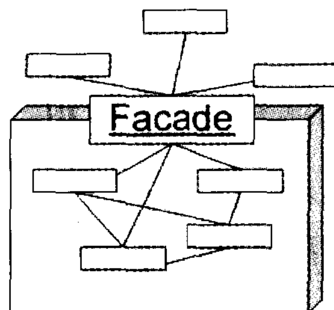
如果一个子系统内部的对象构成自己的"朋友圈"，而子系统外部的对象都属于"陌生人"的话，那么子系统外部的对象与内部的对象就不应当直接通信，而应当通过一个双方都认可的朋友，也就是"门面"对象进行通信，这就导致了门面模式。

下面所示的对象图就显示了一个子系统（一方框代表）内部和外部的通信。可以看出，内部对象与外部对象之间耦合过度，应当降低它们之间的耦合度。



门面模式创造出一个门面对象，将客户端所涉及的属于一个子系统的协作伙伴的数目减到最少，使得客户端与子系统内部的对象的相互作用被门面对象所取代。显然，门面模式就是实现代码重构以便达到迪米特法则要求的一个强有力的武器。

下面所示的对象图就显示了使用门面模式改造之后的情况。

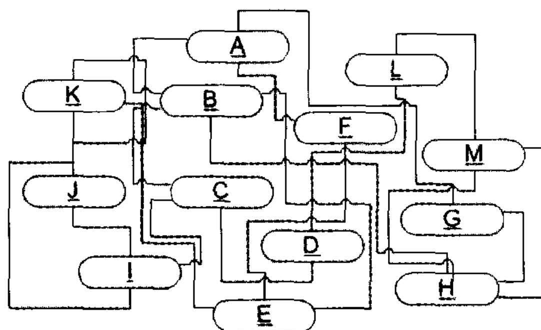


这样做的好处是很显然的，如果修改子系统内部的话，不会直接影响到外部对象；而外部对象的修改不会导致子系统内部的修改。这样一来，子系统内外的演化就相对绝缘了。

关于门面模式，请参见本书“门面（Facade）模式”一章。

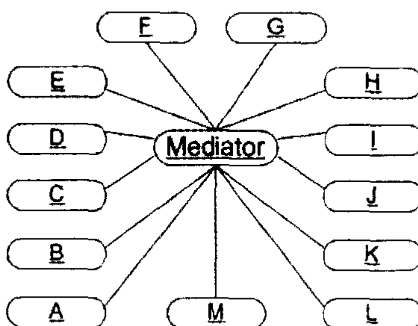
调停者模式

这里一些对象形成一个中等规模的“朋友圈”，在圈内很多的对象都有排列组合般的交互作用，如下图所示。



这时，可以通过创造出一个大家共有的“朋友”对象，然后大家都通过这个“朋友”对象发生相互作用，而将相互之间的直接相互作用省略掉，这就导致了调停者模式。

调停者模式创造出一个调停者对象，将系统中有关的对象所引用的其他对象数目减到最少，使得一个对象与其同事的相互作用被这个对象与调停者对象的相互作用所取代，如下图所示。显然，调停者模式也是迪米特法则的一个具体应用。



关于调停者模式，请参见本书"调停者（Mediator）模式"一章。

由此可见，迪米特法则是贯穿设计模式的核心设计思想，它可以通过设计模式体现出来。而理解了这一思想，也有助于学习设计模式，将设计模式的思想应用到自己的设计中去。

11.4 广义的迪米特法则

其实，迪米特法则所谈论的，就是对对象之间的信息流量、流向以及信息的影响的控制。

在软件系统中，一个模块设计得好不好的最主要、最重要的标志，就是该模块在多大的程度上将自己的内部数据和其他与实现有关的细节隐藏起来。一个设计得好的模块可以将它所有的实现细节隐藏起来，彻底地将提供给外界的 API 和自己的实现分隔开来。这样一来，模块与模块之间就可以仅仅通过彼此的 API 相互通信，而不理会模块内部的工作细节。这一概念就是"信息的隐藏"，或者叫做"封装"，也就是大家熟悉的软件设计的基本教义之一。

信息的隐藏非常重要的原因在于，它可以使各个子系统之间脱耦，从而允许它们独立地被开发、优化、使用、阅读以及修改[BLOCH01]。这种脱耦化可以有效地加快系统的开发过程，因为可以独立地同时开发各个模块。它可以使维护过程变得容易，因为所有的模块都容易读懂，特别是不必担心对其他模块的影响。

虽然信息的隐藏本身并不能带来更好的性能，但是它可以使性能的有效调整变得容易。一旦确认某一个模块是性能的障碍时，设计人员可以针对这个模块本身进行优化，而不必担心影响到其他的模块。

信息的隐藏可以促进软件的复用。由于每一个模块都不依赖于其他模块而存在，因此每一个模块都可以独立地在其他的地方使用。一个系统的规模越大，信息的隐藏就越是重要，而信息隐藏的威力也就越明显。

迪米特法则的主要用意是控制信息的过载。在将迪米特法则运用到系统设计中时，要注意下面的几点：

- 在类的划分上，应当创建有弱耦合的类。类之间的耦合越弱，就越有利于复用。一个处在弱耦合中的类一旦被修改，不会对有关系的类造成波及。
- 在类的结构设计上，每一个类都应当尽量降低成员的访问权限（Accessibility）。换言之，一个类包装好各自的 private 状态（自己的"秘密"）。这样一来，想要了解其中的一个类的意义时，不需要了解很多别的类的细节。一个类不应当 public 自己的属性，而应当提供取值和赋值方法让外界间接访问自己的属性。
- 在类的设计上，只要有可能，一个类应当设计成不变类。
- 在对其他类的引用上，一个对象对其对象的引用应当降到最低。

下面以分类设计和代码层次来讨论迪米特法则的应用问题。



11.5　广义迪米特法则在类的设计上的体现

优先考虑将一个类设置成不变类

Java 语言的 API 中提供了很多的不变类，比如：String，BigInteger，BigDecimal 等封装类都是不变类。不变类易于设计、实现和使用。

一个对象与外界的通信大体可以分成两种，一种是改变这个对象的状态的，另一种是不改变这个对象的状态的。如果一个对象的内部状态根本就是不可能改变的，那么它与外界的通信当然就大大地打了折扣。

当涉及任何一个类的时候，都首先考虑这个类的状态是否需要改变。即便一个类必须是可变类，在给它的属性设置赋值方法的时候，也要保持吝啬的态度。除非真的需要，否则不要为一个属性设置赋值方法。

尽量降低一个类的访问权限

在满足一个系统对这个类的需求的同时，应当尽量降低这个类的访问权限（accessibility）。对于顶级（top-level）的类来说，只有两个可能的访问性等级：

- package-private：这是默认访问权限。如果一个类是 package-private 的，那么它就只能从当前库访问。
- public：如果一个类是 public 的，那么这个类从当前库和其他库都可以访问。

一个类具有 package-private 访问权限的好处是，一旦这个类发生修改，那么受到影响的客户端必定都在这个库内部。由于一个软件包往往有它自己的库结构，因此一个访问权限为 package-private 的类是不会被客户应用程序使用的。这就意味着软件提供商可以自由地决定修改这个类、增加新的 package-private 类，或者删除任何一个 package-private 类，而不必担心对客户的承诺。

相反，如果一个类被不恰当地设置成 public，那么客户程序就有可能会使用这个类。一旦这个类在一个新版本中被删除，就有可能造成一些客户的程序停止运行的情况。

因此，如果一个类可以设置成为 package-private 的，那么就不应当将它设置成为 public 的。

谨慎使用 Serializable

一个类如果实现了 Serializable 接口的话，客户端就可以将这个类的实例串行化，然后再并行化。由于串行化和并行化涉及到类的内部结构，如果这个类的内部 private 结构在一个新版本中发生变化的话，那么客户端可能会根据新版本的结构试图将一个老版本的串行化结果并行化[BLOCH01]，这会导致失败。

换言之，为防止这种情况发生，软件提供商一旦将一个类设置成为 Serializable 的，就不能再在新版本中修改这个类的内部结构，包括 private 的方法和句段。

因此，除非绝对必要，不要使用 Serializable。

在读者阅读本书的“原始模型（Prototype）模式”以及“专题：JavaBean 的‘冷藏’和‘解冻’”等章节时，一定注意不要滥用串行化。

尽量降低成员的访问权限

类的成员包括属性、方法、嵌套类、嵌套接口等，一个类的成员可以有四种不同的访问权限：

- private：这个成员只可能从当前的顶级类的内部访问。
- package-private：这个成员可以被当前库中的任何一个类访问，这是默认访问权限。
- protected：如果一个成员是 protected 的，那么当前库中的任何一个类都可以访问它，而且在任何库中的这个类的子类也都可以访问它。
- public：此成员可以从任何地方被访问。

作为一条指导原则，在设计一个类的方法时，必须首先考虑将其设置成为 private 的[BLOCH01]。只有在发现当前库中还有别的类需要调用这个方法时，才可将其访问权限改为 package-private。如果有太多的方法是 package-private 的，那么就需要检查一下这个类是否划分得当，是不是可以重新设计成两个或者多个类。

对于一个 public 的类来说，将一个方法从 package-private 改成 protected 或者 public，意味着它的访问权限有了巨大的变化[BLOCH01]。一旦一个方法被设置成为 protected，这个方法就可以被位于另一个库中的子类访问；如果设置成 public，那么就可以被任何的类访问。对于一个软件提供商，这就意味着会有客户程序使用这个方法，因此在所有以后的版本中都要承诺不改变这个方法的特征（Signature）。

因此，将 private 或者 package-private 方法改为 protected 或者 public，必须慎之又慎。

取代 C Struct

Java 语言与 C 语言有很多的相似之处，很多 C 程序员认为“Java 是没有指针的 C 语言”。但是相比之下，C 语言是一个面向数据的语言，而 Java 语言是一个面向对象的语言。

C 语言中的 Struct 结构（以及 enum 结构等）就是一个面向数据的构造，在 Java 语言中没有发明一个类似的结构是非常正确的。一个 C 程序员在刚刚使用 Java 编程的时候，往往会认为可以像下面代码清单 5 所示这样实现 C 语言的 Struct 结构。

代码清单 5：一个类似于 C Struct 的 Java 类

```
public class Point
{
    public int x;
    public int y;
```



```
    }
```

这样做在语法上是对的，但是从面向对象设计的角度来说是错误的设计。上面的类 Point 常常叫做退化的类，因为这个类没有提供数据的封装。

一个 C 的 Struct 仅仅是将几个数据段放到一个单一的数据结构中；而一个 Java 类却可以具备方法，从而将方法所涉及的数据向外界隐藏起来。换言之，一个类可以将其数据封装到一个对象中，仅允许通过适当的方法访问这些数据，从而使设计师可以在需要的时候改换这些数据的表象。显然，C 的 Struct 不能提供这些好处。

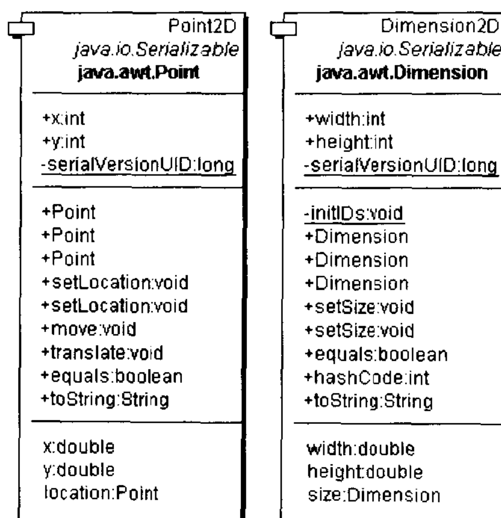
上面的类 Point 常常叫做退化的类，因为这个类没有提供数据的封装。它的设计之所以是错误的，是因为这个类没有提供给自己演化的空间。一旦将来这个类中的数据 *x* 和 *y* 的表象发生变化，必须修改的话，就只好将类的 API 改掉，从而导致所有引用到这个类的其他类做出修改。

一个好的设计应当提供适当的访问方法，包括取值方法和赋值方法，如代码清单 6 所示。

代码清单 6：一个好的设计

```
public class Point
{
    private int x;
    private int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int getX()
    {
        return x;
    }
    public int getY()
    {
        return y;
    }
    public void setX(int x)
    {
        this.x = x;
    }
    public void setY(int y)
    {
        this.y = y;
    }
}
```

读者可能会感到吃惊的是，Java 语言自己的 API 也会犯这种错误[BLOCH01]。下图所示的是 java.awt 库的类图，可以看出，它们都具有 public 的句段。



11.6 广义迪米特法则在代码层次上的实现

限制局域变量的有效范围

C 语言要求所有的局域变量都在一个程序块的开头声明，而 Java 语言允许一个变量在任何地方声明[BLOCH01]，即任何可以有语句的地方都可以声明变量。这样做的意义之深远，是很多人想不到的。

在需要一个变量的时候才声明它，可以有效地限制局域变量的有效范围。一个变量如果仅仅在块的内部使用的话，就应当将这个变量在程序块的内部使用它的地方声明，而不是放到块的外部或者块的开头声明。这样做有两个好处：

（1）程序员可以很容易读懂程序。

设想如果所有的变量都是在开头声明的话，程序员必须反复对照使用变量的语句和变量的声明语句才能将变量的使用与声明对上号。有很多变量在声明后就再也没有使用过，或者随着程序一遍遍地修改，最后有很多变量不再使用了，但是也没有人将声明语句删除。

如果局域变量都是仅仅在马上就要使用的时候才声明，就可以避免上面的情况。

（2）如果一个变量是在需要它的程序块的外部声明的，那么当这个块还没有被执行时，这个变量就已经被分配了内存；而在这个程序块已经执行完毕后，这个变量所占据的内存空间还没有被释放，这显然是不好的。

同样，如果局域变量都是在马上就要使用的时候才声明，也可以避免这种情况。

11.7 老子论“圣人之治”

就中国传统哲学而言，所谓的“道”就是天下的出路，诸子百家，概莫能外。然而，



哲学并不局限于人类社会，哲学可以投放到各种系统中去加以应用，这其中就包括软件系统。

一个软件系统可以看成是一个虚拟的世界，设计师本人就是“驾驭”这个世界的“统治者”。那么中国古代圣贤的智慧完全可以应用到软件系统及其设计中去。

使民无知

《老子》第三章曰：“是以圣人之治，虚其心，实其腹，弱其志，常使民无知无欲。”使被“统治”的对象“愚昧”化，处于“无知”的状态，可以使“统治”的成本降低。

所谓“最少知识”原则，实际上便是老子的“使民无知”的统治之术。

不相往来

《老子》云：“小国寡民……邻国相望，鸡犬之声相闻，民至老死，不相往来。”这里讲的是降低统治成本的方法。将被统治的对象隔离开来，使它们没有直接的通信，可以达到分化瓦解，继而分而治之的效果。

所谓迪米特法则，便是设法使一个软件系统的不同对象彼此之间尽量“老死不相往来”，降低系统维护成本的法则，它与老子的“小国寡民”的统治之术不谋而合。

问答题

1. 从迪米特法则的角度考察下面的两个做法，哪一个更好一些[BLOCH01]，如代码清单 7 和代码清单 8 所示。

代码清单 7：一个 for 循环

```
for (Iterator it = vec.iterator(); it.hasNext())
{
    ...
}
```

代码清单 8：一个 while 循环

```
Iterator it = vec.iterator();
while(it.hasNext())
{
    ...
}
```

2. 从迪米特法则的角度考察下面的两个说法：

   你走你的阳关道，我走我的独木桥。

   井水不犯河水。

3. 从迪米特法则的角度考察下面的说法：

   城门失火，殃及池鱼。

4. 从迪米特法则的角度考察下面的说法：
   四海之内皆兄弟。
5. 从不满足迪米特法则的系统出发，用依赖倒转原则重构这个系统。

问答题答案

1. 第一个更好一些。这是因为变量 it 完全被局限在 for 程序块内部，从而不可能发生被外部意外修改的情况。

因此，一般而言，for 循环比 while 循环稍好一点[BLOCH01]。

2. 这两个说法都是建议减少与外界的联系。这样一旦外界出现变化，不会立即波及到自身。它们都是迪米特法则的同义语。

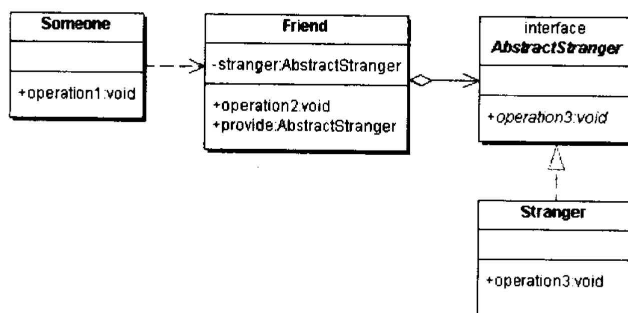
3. 这个说法是从反面说明迪米特法则的重要性。

当城门失火的时候，人们会使用护城河中的水救火，自然就会影响到护城河中的鱼。换言之，如果一个软件实体与另一个实体的关系太过紧密的话，就容易在它出现灾害的时候被波及到。

聪明的鱼应当生活在远离城门的地方，这是来自迪米特法则的建议。

4. 这是违背迪米特法则的。记住，设计师是处在系统中的"兄弟们"的对立面上的。如果系统中的软件模块都像兄弟一样"亲近"，系统的设计就是失败的。

5. 正如前面所指出的，可以通过引进一个抽象的 AbstractStranger 角色，让 Someone 依赖于这个抽象角色，从而使 Someone 与 Stranger 的具体实现脱耦，如下图所示。



改造之后，AbstractStranger 有一个 Java 接口实现，如代码清单 9 所示。

代码清单 9：AbstractStranger 接口的源代码

```
public interface AbstractStranger
{
    abstract void operation3();
}
```

Stranger 可以实现这个接口，从而使 Stranger 具有一个抽象类型 AbstractStranger。Someone 则依赖于这个抽象类型，如代码清单 10 所示。

代码清单 10：Friend 类的源代码

```
package com.javapatterns.lod.forwarding3;
public class Someone
```

```
{
    public void operation1(Friend friend)
    {
        AbstractStranger stranger = friend.provide();
        stranger.operation3();
    }
}
```

同样地，Friend 所提供的也是这个抽象类型，如代码清单 11 所示。

代码清单 11：Friend 类的源代码

```
package com.javapatterns.lod.forwarding3;
public class Friend
{
    private AbstractStranger stranger = new Stranger();
    public void operation2()
    {
        System.out.println("In Friend.operation2()");
    }
    public AbstractStranger provide()
    {
        return stranger;
    }
}
```

具体类 Stranger 的源代码被省略。

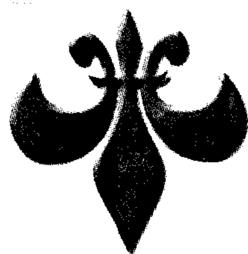
参考文献

[LIEB89] Karl. J. Lieberherr and I. Holland. Assuring good style for object-oriented programs. IEEE Software，September 1989，P38～48

[LIEB96] Karl Lieberherr. Adaptive Object Oriented Software–The Demeter Method. PWS Publishing Co.，Boston，1996

[KIRK02] Kirk Knoernschild. Java Design–Objects，UML，and Process. Addison-Wesley，2002

[BLOCH01] Joshua Bloch. Effective Java–Programming Language Guide. published by Addison-Wesley，2001



第三部分 创建模式

创建模式（Creational Pattern）是对类的实例化过程的抽象化。一些系统在创建对象时，需要动态地决定怎样创建对象，创建哪些对象，以及如何组合和表示这些对象。创建模式描述了怎样构造和封装这些动态的决定。

创建模式分为类的创建模式和对象的创建模式两种。

- 类的创建模式　类的创建模式使用继承关系，把类的创建延迟到子类，从而封装了客户端将得到哪些具体类的信息，并且隐藏了这些类的实例是如何被创建和放在一起的。
- 对象的创建模式　而对象的创建模式则把对象的创建过程动态地委派给另一个对象，从而动态地决定客户端将得到哪些具体类的实例，以及这些类的实例是如何被创建和组合在一起的。

本书将要介绍的创建模式包括以下几种：简单工厂模式、工厂方法模式、抽象工厂模式、单例模式、多例模式、建造模式、原始模型模式等。

此外，与单例模式一同介绍的，还有“专题：单例模式与 MX 记录”；与多例模式一同介绍的，还有“专题：序列键生成器与单例及多例模式”；与原始模型模式一同介绍的，还有“专题：JavaBean 的‘冷藏’和‘解冻’”。

第 12 章　简单工厂（Simple Factory）模式

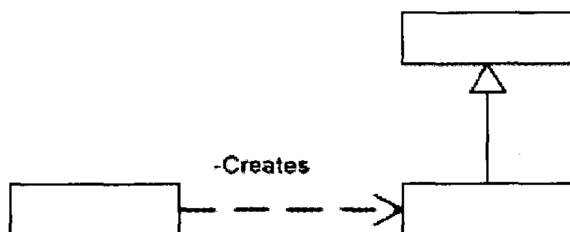
简单工厂模式是类的创建模式，又叫做静态工厂方法（Static Factory Method）模式。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。

12.1　工厂模式的几种形态

工厂模式专门负责将大量有共同接口的类实例化。工厂模式可以动态决定将哪一个类实例化，不必事先知道每次要实例化哪一个类。工厂模式有以下几种形态：

- 简单工厂（Simple Factory）模式：又称静态工厂方法模式（Static Factory Method Pattern）。
- 工厂方法（Factory Method）模式：又称多态性工厂（Polymorphic Factory）模式或虚拟构造子（Virtual Constructor）模式。
- 抽象工厂（Abstract Factory）模式：又称工具箱（Kit 或 Toolkit）模式。

下图所示就是简单工厂模式的简略类图。



简单工厂模式，或称静态工厂方法模式，是不同的工厂方法模式的一个特殊实现。在其他文献中，简单工厂模式往往作为普通工厂模式的一个特例讨论。

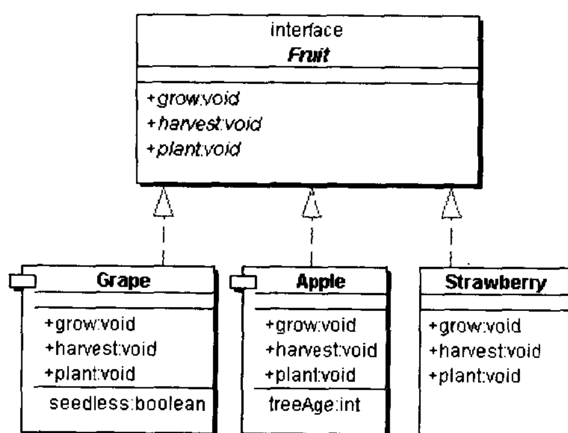
在 Java 语言中，通常的工厂方法模式不能通过设计功能的退化给出静态工厂方法模式。因为一个方法是不是静态的，对于 Java 语言来说是一个很大的区别，必须在一开始就加以考虑。这就是本书将简单工厂单独提出来讨论的一个原因。学习简单工厂模式是对学习工厂方法模式的一个很好的准备，也是对学习其他模式，特别是单例模式和多例模式的一个很好的准备，这就是本书首先讲解这一模式的另一个原因。

12.2 简单工厂模式的引进

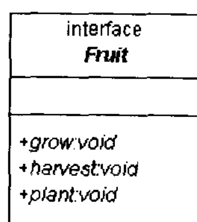
比如说有一个农场公司，专门向市场销售各类水果。在这个系统里需要描述下列的水果：

- 葡萄 Grape
- 草莓 Strawberry
- 苹果 Apple

水果与其他的植物有很大的不同，水果最终是可以采摘食用的。那么一个自然的作法就是建立一个各种水果都适用的接口，以便与农场里的其他植物区分开，如下图所示。



水果接口规定出所有的水果必须实现的接口，包括任何水果类必须具备的方法：种植 plant()，生长 grow()以及收获 harvest()。接口 Fruit 的类图如下图所示。



这个水果接口的源代码如代码清单 1 所示。

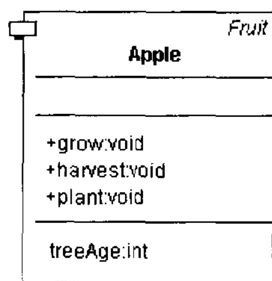
代码清单 1：接口 Fruit 的源代码

```
public interface Fruit
{
    /**
     *  生长
     */
    void grow();
    /**
```



```
     *  收获
     */
     void harvest();
    /**
     *  种植
     */
     void plant();
}
```

描述苹果的 Apple 类的类图如下图所示。



Apple 类是水果类的一种，因此它实现了水果接口所声明的所有方法。另外，由于苹果是多年生植物，因此多出一个 treeAge 性质，描述苹果树的树龄。下面是这个 Apple 类的源代码，如代码清单 2 所示。

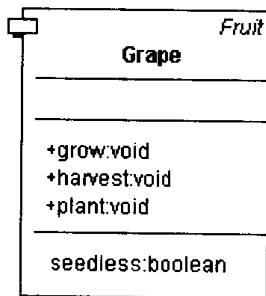
代码清单 2：　Apple 类的源代码

```
public class Apple implements Fruit
{
    private int treeAge;

    /**
     *  生长
     */
    public void grow()
    {
        log("Apple is growing...");
    }
    /**
     *  收获
     */
    public void harvest()
    {
        log("Apple has been harvested.");
    }
    /**
     *  种植
     */
    public void plant()
```

```
        {
            log("Apple has been planted.");
        }
        /**
         *  辅助方法
         */
        public static void log(String msg)
        {
            System.out.println(msg);
        }
        /**
         *  树龄的取值方法
         */
        public int getTreeAge()
        {
            return treeAge;
        }
        /**
         *  树龄的赋值方法
         */
        public void setTreeAge(int treeAge)
        {
            this.treeAge = treeAge;
        }
    }
```

同样，Grape 类是水果类的一种，也实现了 Fruit 接口所声明的所有的方法。但由于葡萄分有籽和无籽两种，因此，比通常的水果多出一个 seedless 性质，如下图所示。



葡萄类的源代码如代码清单 3 所示。可以看出，Grape 类同样实现了水果接口，从而是水果类型的一种子类型。

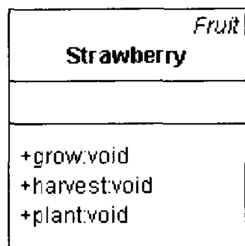
代码清单 3： Grape 类的源代码

```
public class Grape implements Fruit
{
    private boolean seedless;

    /**
```

```
     *  生长
     */
    public void grow()
    {
        log("Grape is growing...");
    }
    /**
     *  收获
     */
    public void harvest()
    {
        log("Grape has been harvested.");
    }
    /**
     *  种植
     */
    public void plant()
    {
        log("Grape has been planted.");
    }
    /**
     *  辅助方法
     */
    public static void log(String msg)
    {
        System.out.println(msg);
    }
    /**
     *  有无籽的取值方法
     */
    public boolean getSeedless()
    {
        return seedless;
    }
    /**
     *  有无籽的赋值方法
     */
    public void setSeedless(boolean seedless)
    {
        this.seedless = seedless;
    }
}
```

下图所示是 Strawberry 类的类图。



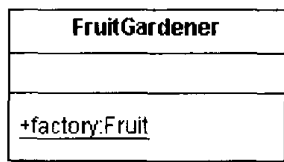
Strawberry 类实现了 Fruit 接口，因此，也是水果类型的子类型，其源代码如代码清单 4 所示。

代码清单 4：类 Strawberry 的源代码

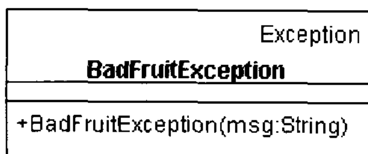
```
public class Strawberry implements Fruit
{
    /**
     *  生长
     */
    public void grow()
    {
        log("Strawberry is growing...");
    }
    /**
     *  收获
     */
    public void harvest()
    {
        log("Strawberry has been harvested.");
    }
    /**
     *  种植
     */
    public void plant()
    {
        log("Strawberry has been planted.");
    }
    /**
     *  辅助方法
     */
    public static void log(String msg)
    {
        System.out.println(msg);
    }
}
```

农场的园丁也是系统的一部分，自然要由一个合适的类来代表。这个类就是

FruitGardener 类，其结构图如下图所示。



FruitGardener 类会根据客户端的要求，创建出不同的水果对象，比如苹果（Apple），葡萄（Grape）或草莓（Strawberry）的实例。而如果接到不合法的要求，FruitGardener 类会抛出 BadFruitException 异常，如下图所示。



园丁类的源代码如代码清单 5 所示。

代码清单 5：FruitGardener 类的源代码

```
public class FruitGardener
{
    /**
     * 静态工厂方法
     */
    public static Fruit factory(String which)
            throws BadFruitException
    {
        if (which.equalsIgnoreCase("apple"))
        {
            return new Apple();
        }
        else if (which.equalsIgnoreCase("strawberry"))
        {
            return new Strawberry();
        }
        else if (which.equalsIgnoreCase("grape"))
        {
            return new Grape();
        }
        else
        {
            throw new BadFruitException("Bad fruit request");
        }
    }
}
```

可以看出，园丁类提供了一个静态工厂方法。在客户端的调用下，这个方法创建客户端所需要的水果对象。如果客户端的请求是系统所不支持的，工厂方法就会抛出一个 BadFruitException 异常。这个异常类的源代码如代码清单 6 所示。

代码清单 6：BadFruitException 类的源代码

```
public class BadFruitException extends Exception
{
    public BadFruitException(String msg)
    {
        super(msg);
    }
}
```

在使用时，客户端只需调用 FruitGardener 的静态方法 factory()即可。请见下面的示意性客户端源代码，如代码清单 7 所示。

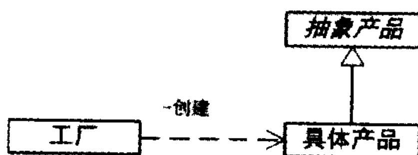
代码清单 7：怎样使用异常类 BadFruitException

```
try
{
    FruitGardener.factory("grape");
    FruitGardener.factory("apple");
    FruitGardener.factory("strawberry");
        ...
}
catch(BadFruitException e)
{
    ...
}
```

这样，农场一定会百果丰收啦！

12.3 简单工厂模式的结构

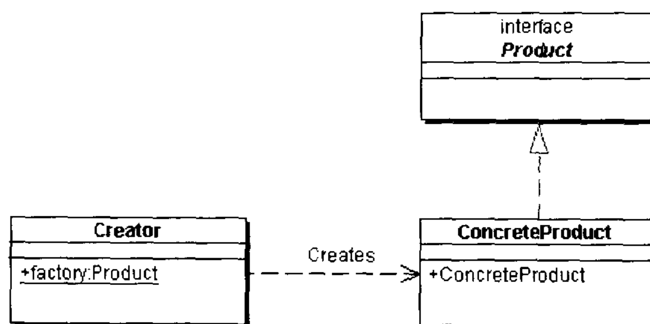
简单工厂模式是类的创建模式，这个模式的一般性结构如下图所示。



角色与结构

简单工厂模式就是由一个工厂类根据传入的参量决定创建出哪一种产品类的实例。下

面以一个示意性的实现为例说明简单工厂模式的结构，如下图所示。



从上图可以看出，简单工厂模式涉及到工厂角色、抽象产品角色以及具体产品角色等三个角色：

- 工厂类（Creator）角色：担任这个角色的是工厂方法模式的核心，含有与应用紧密相关的商业逻辑。工厂类在客户端的直接调用下创建产品对象，它往往由一个具体 Java 类实现。
- 抽象产品（Product）角色：担任这个角色的类是由工厂方法模式所创建的对象的父类，或它们共同拥有的接口。抽象产品角色可以用一个 Java 接口或者 Java 抽象类实现。
- 具体产品（Concrete Product）角色：工厂方法模式所创建的任何对象都是这个角色的实例，具体产品角色由一个具体 Java 类实现。

源代码

工厂类的示意性源代码如代码清单 8 所示。可以看出，这个工厂方法创建了一个新的具体产品的实例并返还给调用者。

代码清单 8：Creator 类的源代码

```
public class Creator
{
    /**
     *  静态工厂方法
     */
    public static Product factory()
    {
        return new ConcreteProduct();
    }
}
```

抽象产品角色的主要目的是给所有的具体产品类提供一个共同的类型，在最简单的情况下，可以简化为一个标识接口，如代码清单 9 所示。所谓标识接口，就是没有声明任何方法的空接口。关于标识接口的讨论，请参见本书的“专题：Java 接口”一章。

代码清单 9：抽象角色 Product 接口的源代码

```
public interface Product
{
}
```

具体产品类的示意性源代码如代码清单 10 所示。

代码清单 10：具体产品角色 ConcreteProduct 类的源代码

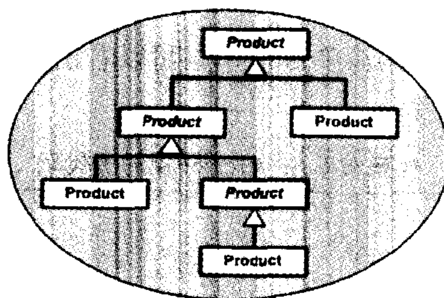
```
public class ConcreteProduct implements Product
{
    public ConcreteProduct(){}
}
```

虽然在这个简单的示意性实现里面只给出了一个具体产品类，但是在实际应用中一般都会遇到多个具体产品类的情况。

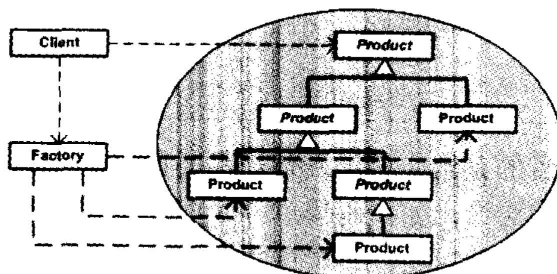
12.4 简单工厂模式的实现

多层次的产品结构

在真实的系统中，产品可以形成复杂的等级结构，比如下图所示的树状结构上就有多个抽象产品类和具体产品类。



这个时候，简单工厂模式采取的是以不变应万变的策略，一律使用同一个工厂类，如下图所示。



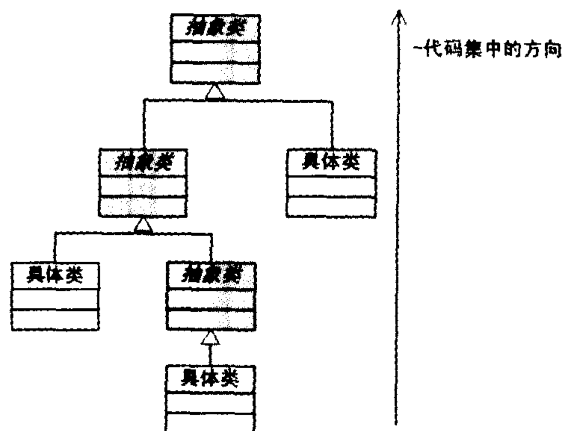


图中从 Factory 类到各个 Product 类的虚线代表创建（依赖）关系；从 Client 到其他类的连线是一般依赖关系。

这样做的好处是设计简单，产品类的等级结构不会反映到工厂类中来，从而产品类的等级结构的变化也就不会影响到工厂类。但是这样做的缺点是，增加新的产品必将导致工厂类的修改。请参见后面“简单工厂模式的优点和缺点”一节，以及本书的“工厂方法(Factory Method）模式”一章。

使用 Java 接口或者 Java 抽象类

如果模式所产生的具体产品类彼此之间没有共同的商业逻辑，那么抽象产品角色可以由一个 Java 接口扮演；相反，如果这些具体产品类彼此之间确实有共同的商业逻辑，那么这些公有的逻辑就应当移到抽象角色里面，这就意味着抽象角色应当由一个抽象类扮演。在一个类型的等级结构里面，共同的代码应当尽量向上移动，以达到共享的目的，如下图所示。



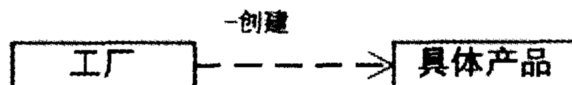
关于抽象类与 Java 接口的关系与区别，请参见本书的“专题：Java 接口”与“专题：抽象类”两章。

多个工厂方法

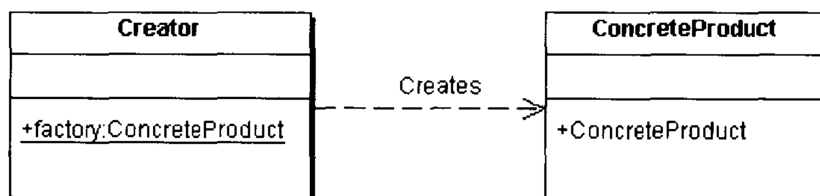
每个工厂类可以有多于一个的工厂方法，分别负责创建不同的产品对象。比如 java.text.DateFormat 类是其子类的工厂类，而 DateFormat 类就提供了多个静态工厂方法。请参见本章后面的详尽讲解。

抽象产品角色的省略

如果系统仅有一个具体产品角色的话，那么就可以省略掉抽象产品角色。省略掉抽象产品类后的简略类图如下图所示。



仍然以前面给出的示意性系统为例，这时候系统的类图如下图所示。



工厂类的源代码如代码清单 11 所示。显然，这个类提供一个工厂方法，返还一个具体产品类的实例。

代码清单 11：工厂角色的源代码

```
package com.javapatterns.simplefactory.simplified;
public class Creator
{
    /**
     *  静态工厂方法
     */
    public static ConcreteProduct factory()
    {
        return new ConcreteProduct();
    }
}
```

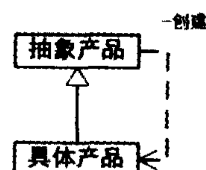
具体产品类的源代码如代码清单 12 所示。

代码清单 12：具体产品角色 ConcreteProduct 类的源代码

```
package com.javapatterns.simplefactory.simplified;
public class ConcreteProduct
{
    public ConcreteProduct(){}
}
```

工厂角色与抽象产品角色合并

在有些情况下，工厂角色可以由抽象产品角色扮演。典型的应用就是 java.text.DateFormat 类，一个抽象产品类同时是子类的工厂，如右图所示。



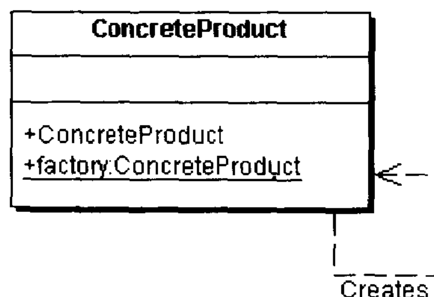
本章在下面给出了一个更为详尽的描述。

三个角色全部合并

如果抽象产品角色已经被省略，而工厂角色就可以与具体产品角色合并。换言之，一个产品类为自身的工厂，如右图所示。

-创建
具体产品

如果仍然以前面讨论过的示意性系统为例，这个系统的结构图如下图所示。



显然，三个原本独立的角色：工厂角色、抽象产品以及具体产品角色都已经合并成为一个类，这个类自行创建自己的实例，如代码清单 13 所示。

代码清单 13：具体产品角色与工厂角色合并后的源代码

```
package com.javapatterns.simplefactory.simplified1;
public class ConcreteProduct
{
    public ConcreteProduct(){}
   /**
    *  静态工厂方法
    */
    public static ConcreteProduct factory()
    {
        return new ConcreteProduct();
    }
}
```

这种退化的简单工厂模式与单例模式以及多例模式有相似之处，但是并不等于单例或者多例模式。关于这一点的讨论请参见本章后面的叙述。

产品对象的循环使用和登记式的工厂方法

由于简单工厂模式是一个非常基本的设计模式，因此它会在较为复杂的设计模式中出现。在本章前面所给出的示意性例子中，工厂方法总是简单地调用产品类的构造子以创建一个新的产品实例，然后将这个实例提供给客户端，而在实际情形中工厂方法所做的事情可以相当复杂。

在本书所讨论的所有设计模式中，单例模式与多例模式是建立在简单工厂模式的基础

之上的，而且它们都要求工厂方法具有特殊的逻辑，以便能循环使用产品的实例。

在很多情况下，产品对象可以循环使用。换言之，工厂方法可以循环使用已经创建出来的对象，而不是每一次都创建新的产品对象。工厂方法可以通过登记它所创建的产品对象来达到循环使用产品对象的目的。

如果工厂方法总是循环使用同一个产品对象，那么这个工厂对象可以使用一个属性来存储这个产品对象。每一次客户端调用工厂方法时，工厂方法总是提供这同一个对象。在单例模式中就是这样，单例类提供一个静态工厂方法，向外界提供一个惟一的单例类实例。

如果工厂方法永远循环使用固定数目的一些产品对象，而且这些产品对象的数目并不大的话，可以使用一些私有属性存储对这些产品对象的引用。比如，一个永远只提供一个产品对象的工厂对象可以使用一个静态变量存储对这个产品对象的引用。

相反，如果工厂方法使用数目不确定，或者数目较大的一些产品对象的话，使用属性变量存储对这些产品对象的引用就不方便了。这时候，就应当使用聚集对象存储对产品对象的引用。

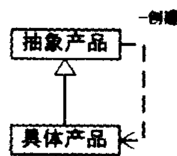
不管使用哪一种方法，工厂方法都可以做到循环使用它所创建的产品对象。循环的逻辑可能是基于这些产品类的内部状态，比如某一种状态的产品对象只创建一个，让所有需要处于这一状态上的产品对象的客户端共享这一个实例。

请参见下面对单例模式和多例模式的讨论。

12.5 简单工厂模式与其他模式的关系

单例模式

单例模式使用了简单工厂模式。换言之，单例类具有一个静态工厂方法提供自身的实例。一个抽象产品类同时是子类的工厂，如右图所示。



但是单例模式并不是简单工厂模式的退化情形，单例模式要求单例类的构造子是私有的，从而客户端不能直接将之实例化，而必须通过这个静态工厂方法将之实例化，而且单例类自身是自己的工厂角色。换言之，单例类自己负责创建自身的实例。

单例类使用一个静态的属性存储自己的惟一实例，工厂方法永远仅提供这一个实例。

多例模式

多例模式是对单例模式的推广。多例模式与单例模式的共同之处在于它们都禁止外界直接将之实例化，同时通过静态工厂方法向外界提供循环使用的自身的实例。它们的不同在于单例模式仅有一个实例，而多例模式则可以有多个实例。

多例模式往往具有一个聚集属性，通过向这个聚集属性登记已经创建过的实例达到循环使用实例的目的。一般而言，一个典型的多例类具有某种内部状态，这个内部状态可以

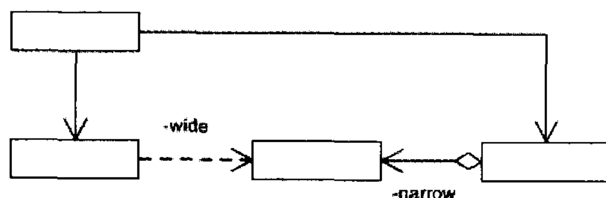
用来区分各个实例，而对应于每一个内部状态，都只有一个实例存在。

根据外界传入的参量，工厂方法可以查询自己的登记聚集，如果具有这个状态的实例已经存在，就直接将这个实例提供给外界；反之，就首先创建一个新的满足要求的实例，将之登记到聚集中，然后再提供给客户端。

关于单例模式和多例模式请读者参阅本书的“单例（Singleton）模式”一章和“多例（Multiton）模式”一章。

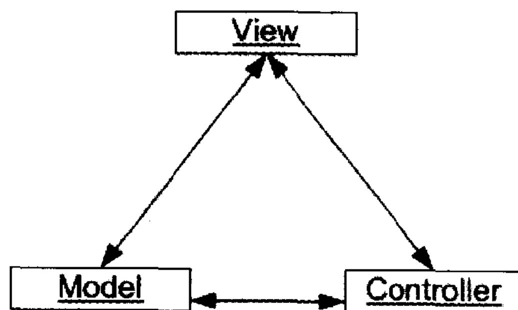
备忘录模式

单例模式和多例模式使用一个属性或者聚集属性来登记所创建的产品对象，以便可以通过查询这个属性或者聚集属性找到并共享已经创建了的产品对象，这就是备忘录模式的应用。备忘录模式的简略类图如下图所示。

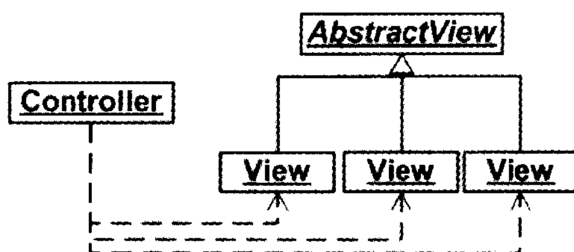


MVC 模式

MVC 模式并不是严格意义上的设计模式，而是在更高层次上的架构模式。MVC 模式可以分解成为几个设计模式的组合，包括合成模式、策略模式、观察者模式，也有可能会包括装饰模式、调停者模式、迭代子模式以及工厂方法模式等。MVC 模式的结构图如下图所示。关于 MVC 模式的讨论可以参考本书的“专题：MVC 模式与用户输入数据检查”一章。



简单工厂模式所创建的对象往往属于一个产品等级结构，这个等级结构可以是 MVC 模式中的视图（View），而工厂角色本身可以是控制器（Controller）。一个 MVC 模式可以有一个控制器和多个视图，如下图所示。



换言之，控制器端可以创建合适的视图端，就如同工厂角色创建合适的对象角色一样；而模型端则可以充当这个创建过程的客户端。

如果系统需要有多个控制器参与这个过程的话，简单工厂模式就不适合了，应当考虑使用工厂方法模式。请参阅本书的“工厂方法（Factory Method）模式”一章。

12.6 简单工厂模式的优点和缺点

简单工厂模式的优点

模式的核心是工厂类。这个类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例。而客户端则可以免除直接创建产品对象的责任，而仅仅负责“消费”产品。简单工厂模式通过这种做法实现了对责任的分割。

简单工厂模式的缺点

正如同在本章前面所讨论的，当产品类有复杂的多层次等级结构时，工厂类只有它自己。以不变应万变，就是模式的缺点。

这个工厂类集中了所有的产品创建逻辑，形成一个无所不知的全能类，有人把这种类叫做上帝类（God Class）。如果这个全能类代表的是农场的一个具体园丁的话，那么这个园丁就需要对所有的产品负责，成了农场的关键人物，他什么时候不能正常工作了，整个农场都要受到影响。

将这么多的逻辑集中放在一个类里面的另外一个缺点是，当产品类有不同的接口种类时，工厂类需要判断在什么时候创建某种产品。这种对时机的判断和对哪一种具体产品的判断逻辑混合在一起，使得系统在将来进行功能扩展时较为困难。这一缺点在工厂方法模式中得到克服。

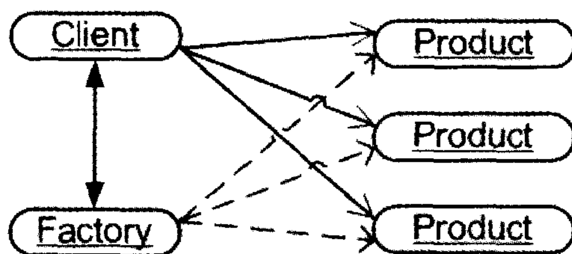
由于简单工厂模式使用静态方法作为工厂方法，而静态方法无法由子类继承，因此，工厂角色无法形成基于继承的等级结构。这一缺点会在工厂方法模式中得到克服。

“开–闭”原则

“开–闭”原则要求一个系统的设计能够允许系统在无需修改的情况下，扩展其功能。

那么简单工厂模式是否满足这个条件呢？

要回答这个问题，首先需要将系统划分成不同的子系统，再考虑功能扩展对于这些子系统的要求。一般而言，一个系统总是可以划分成为产品的消费者角色（Client）、产品的工厂角色（Factory）以及产品角色（Product）三个子系统，如下图所示。



在这个系统中，功能的扩展体现在引进新的产品上。“开–闭”原则要求系统允许当新的产品加入系统中时，而无需对现有代码进行修改。这一点对于产品的消费角色是成立的，而对于工厂角色是不成立的。

对于产品消费角色来说，任何时候需要某种产品，只需向工厂角色请求即可。而工厂角色在接到请求后，会自行判断创建和提供哪一个产品。所以，产品消费角色无需知道它得到的是哪一个产品；换言之，产品消费角色无需修改就可以接纳新的产品。

对于工厂角色来说，增加新的产品是一个痛苦的过程。工厂角色必须知道每一种产品，如何创建它们，以及何时向客户端提供它们。换言之，接纳新的产品意味着修改这个工厂角色的源代码。

综合本节的讨论，简单工厂角色只在有限的程度上支持“开–闭”原则。

12.7 简单工厂模式在 Java 中的应用

简单工厂模式是一个很基本的设计模式，读者可以在 Java 语言的 API 中看到这个模式的应用。下面就以几个 Java API 中的应用为例讲解怎样实现简单工厂模式。

DateFormat 与简单工厂模式

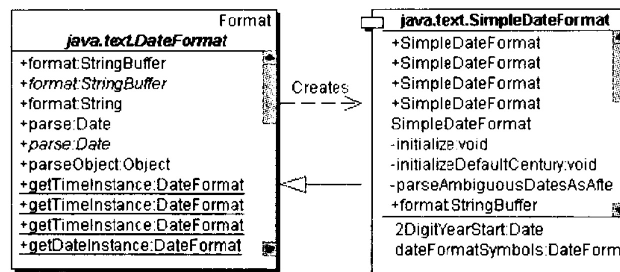
想必大多数的读者都曾经使用工具类 java.text.DateFormat 或其子类来格式化一个本地日期或者时间，这个工具类在处理英语和非英语的日期及时间格式上很有用处。

使用的目的

在本地机器上，时间和日期的格式存在一些标准的风格，比如一个日期可以被格式化为 FULL、LONG、MEDIUM 或者 SHORT，但是它们所代表的意义随着实现的不同而不同。

以英语为例，SHORT 代表完全是数字型的短格式，例如“1/20/2002”或者“15:20”；而 MEDIUM 代表的中格式则是把缩写文字增加到短格式中，比如：“Jul 22, 2002”或者

"3:20pm"；而 LONG 代表的长格式用的是完整的词，比如："July 2, 2002"或者"3:20:10pm"；而 FULL 所代表的全格式包括了完整的日期信息，比如："Monday, July 22, 2002, AD"或者"3:20:10pm EST"等。DateFormat 与 SimpleDateFormat 的类图如下图所示。



getDateInstance()方法

如果仔细考察这个 DateFormat 类就会发现，它是一个抽象类，但是却提供了很多的静态工厂方法，比如 getDateInstance()为某种本地日期提供格式化，它由三个重载的方法组成，如代码清单 14 所示。

代码清单 14：DateFormat 的部分源代码

```
public final static DateFormat getDateInstance();
public final static DateFormat getDateInstance(int style);
public final static DateFormat getDateInstance(int style, Locale locale);
```

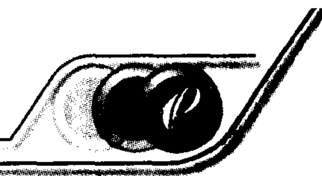
第一次接触这个类的初级程序员会有一些困惑，比如有的人会问，为什么一个抽象类可以有自己的实例，并通过几个方法提供自己的实例。实际上，一个抽象类不能有自己的实例，这是绝对正确的。而应当注意的是，DateFormat 的工厂方法是静态方法，并不是普通的方法。

getDateInstance()方法并没有调用 DateFormat 的构造子来提供自己的实例，作为一个工厂方法，getDateInstance()方法做了一些有趣的事情。它所做的事情基本上可以分成两部分：一是运用多态性；二是使用静态工厂方法。

从上面给出的 DateFormat 和 SimpleDateFormat 的类图可以看出，SimpleDateFormat 是抽象类 DateFormat 的具体子类，这就意味着 SimpleDateFormat 是一个 DateFormat 类型的子类型；而 getDateInstance()方法完全可以返回 SimpleDateFormat 的实例，并且仅将它声明为 DateFormat 类型。这就是最纯正的多态性原则的运用[SINTES00]。

getDateInstance()方法是一个静态方法。如果它是一个非静态的普通方法会怎样呢？要使用这个（非静态）方法，客户端必须首先取得这个类的实例或者其子类的实例。而这个类是一个抽象类，不可能有自己的实例，所以客户端就只好首先取得其具体子类的实例。如果客户端能够取得它的子类的实例，那么还需要这个工厂方法干什么呢？

显然，这里使用静态工厂方法是为了将具体子类实例化的工作隐藏起来，从而客户端不必考虑如何将具体子类实例化，因为抽象类 DateFormat 会提供它的合适的具体子类的实例。这是一个简单工厂方法模式的绝佳应用。



针对抽象编程

这样做是利用具体产品类的超类类型将它的真实类型隐藏起来，其好处是提供了系统的可扩展性。如果将来有新的具体子类被加入到系统中来，那么工厂类可以将交给客户端的对象换成新的子类的实例，而对客户端没有任何影响。

这种将工厂方法的返还类型设置成抽象产品类型的做法，叫做针对抽象编程，这是依赖倒转原则（DIP）的应用。关于这一设计原则的详细讨论，请参见本书的"依赖倒转原则（DIP）"一章。

本地时间

与本地日期的格式化相对应的是为某种本地时间提供格式化，这一功能由三个重载的 getTimeInstance()方法提供，如代码清单 15 所示。

代码清单 15：DateFormat 的部分源代码

```
public final static DateFormat getTimeInstance();
public final static DateFormat getTimeInstance(int style);
public final static DateFormat getTimeInstance(int style, Locale locale);
```

显然它们所提供的也是其具体子类的实例，而不是自身的实例。因为它自身是一个抽象类，不可能有自己的实例。由于其子类必然是 DateFormat 的子类型，因此返还类型可以是 DateFormat 类型，这也是多态性的体现。

一个法语日期的例子

为了进一步说明这个工具类的使用办法，在下面给出的例子中，假定本地语言是法语，并针对法语进行时间和日期的格式化，如代码清单 16 所示。

代码清单 16：DateTester 的部分源代码

```
package com.javapatterns.simplefactory.dateformat;
import java.util.Locale;
import java.util.Date;
import java.text.DateFormat;
import java.text.ParseException;
public class DateTester
{
    public static void main(String[] args)
    {
        Locale locale = Locale.FRENCH;
        Date date = new Date();
        String now = DateFormat.getTimeInstance(DateFormat.DEFAULT, locale)
            .format(date);
        System.out.println(now);
        try
        {
            date = DateFormat.getDateInstance(DateFormat.DEFAULT, locale)
                .parse("16 nov. 01");
            System.out.println(date);
```

```
            }
            catch(ParseException e)
            {
                System.out.println("Parsing exception:"+e);
            }
        }
    }
```

其中 now 包含了按照法语格式写出的当前时间，而 date 则读入了以法语方式书写的一个日期“16 nov. 01”。系统运行时会打印出下面的结果，如代码清单 17 所示。

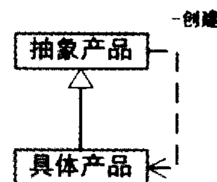
代码清单 17：DateTester 的运行结果

```
13:18:36
Fri Nov 16 00:00:00 PST 2001
```

简单工厂模式的应用

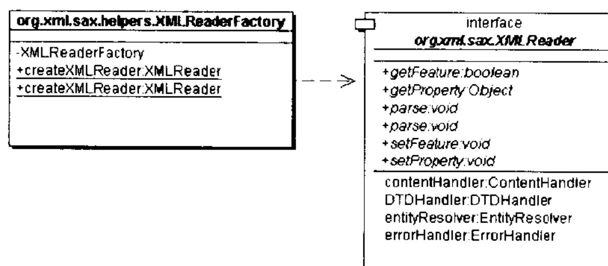
从上面的例子可以看出，由于使用了简单工厂模式，客户端完全不必操心工厂方法所返还的对象是怎样创建和构成的。工厂方法将实例化哪些对象以及如何实例化这些对象的细节隐藏起来，使得对这些对象的使用得到简化。

与一般的简单工厂模式不同的地方在于，这里的工厂角色与抽象产品角色合并成一个类。换言之，抽象产品角色负责具体产品角色的创建，这是简单工厂模式的一个特例。如右图所示。



SAX2 库中的 XMLReaderFactory 与简单工厂模式

在 SAX2 库中，XMLReaderFactory 类使用了简单工厂模式，用来创建产品类 XMLReader 的实例。下图所示是 XMLReaderFactory 和 XMLReader 类型的关系图。



XMLReaderFactory 提供了两种不同的静态方法，适用于不同的驱动软件参数。

关于 SAX2 库的知识以及 SAX2 库所涉及到的其他模式的讨论，请阅读本书“专题：XMLProperties 与适配器模式”和“专题：观察者模式与 SAX2 浏览器”两章。

12.8 女娲抟土造人

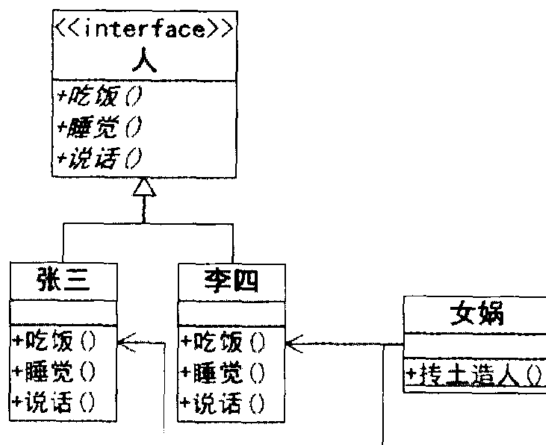
《风俗通》中说：“俗说天地开辟，未有人民。女娲抟黄土为人。”女娲需要用土造出

一个个的人，这就是简单工厂模式的应用。

本章试图使用 UML 和模式的语言来解释女娲的做法。首先，在这个造人的思想里面，有几个重要的角色：女娲本身、抽象的人的概念和女娲所造出的具体的人。

- 女娲是一个工厂类，也就是简单工厂模式的核心角色。
- 具体的一个个的人，包括张三、李四等。这些人便是简单工厂模式里面的具体产品角色。
- 抽象的人便是最早只存在于女娲的头脑里的一个想法，女娲按照这个想法造出的一个一个具体的人便都符合这个抽象的人的定义。换言之，这个抽象的想法规定了所有具体的人必须具有的接口。

女娲抟土造人的 UML 类图如下图所示。



问答题

1. 在本节开始时不是说：工厂模式就是在不使用 new 操作符的情况下，将类实例化的吗？可为什么在具体实现时仍然使用了 new 操作符呢？

2. 请使用简单工厂模式设计一个创建不同几何形状，如圆形、方形和三角形实例的描图员（Art Tracer）系统。每个几何图形都要有画出 draw()和擦去 erase()两个方法。当描图员接到指令，要求创建不支持的几何图形时，要提出 BadShapeException 异常。

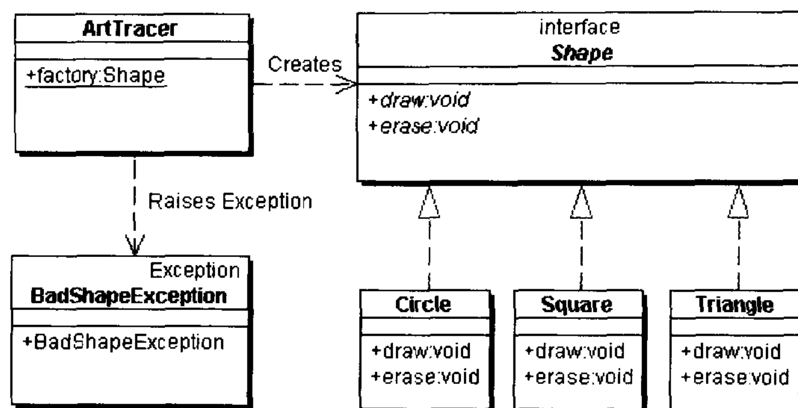
3. 请给出上一题的源代码。

4. 请简单举例说明描图员系统怎样使用。

问答题答案

1. 对整个系统而言，工厂模式把具体使用 new 操作符的细节包装和隐藏起来。当然只要程序是用 Java 语言写的，Java 语言的特征在细节里就一定会出现。

2. 这里给出问题的完整答案。描图员（Art Tracer）系统的 UML 如下图所示。

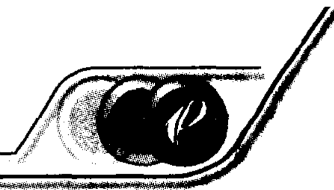


3. 描图员系统包括一个接口扮演抽象产品角色，三个具体产品类，一个工厂类和一个 Exception 类。工厂类便是 ArtTracer 类，它的源代码如代码清单 18 所示。

代码清单 18：ArtTracer 类的源代码

```
public class ArtTracer
{
    /**
     *  静态工厂方法
     */
    public static Shape factory(String which) throws BadShapeException
    {
        if (which.equalsIgnoreCase("circle"))
        {
            return new Circle();
        }
        else if (which.equalsIgnoreCase("square"))
        {
            return new Square();
        }
        else if (which.equalsIgnoreCase("triangle"))
        {
            return new Triangle();
        }
        else
        {
            throw new BadShapeException(which);
        }
    }
}
```

Shape 是一个 Java 接口，规范出所有的产品类必须实现的方法，它的源代码如代码清单 19 所示。



代码清单 19：Shape 接口的源代码

```
public interface Shape
{
    void draw();
    void erase();
}
```

Square 是一个具体产品类，实现抽象产品角色，即 Shape 接口，如代码清单 20 所示。

代码清单 20：Square 类的源代码

```
public class Square implements Shape
{
    public void draw()
    {
        System.out.println("Square.draw()");
    }
    public void erase()
    {
        System.out.println("Square.erase()");
    }
}
```

与上面的 Square 一样，Circle 也是一个具体产品类，实现 Shape 接口，如代码清单 21 所示。

代码清单 21：Circle 类的源代码

```
public class Circle implements Shape
{
    public void draw()
    {
        System.out.println("Circle.draw()");
    }
    public void erase()
    {
        System.out.println("Circle.erase()");
    }
}
```

Triangle 是一个具体产品类，与上面的两个具体产品类一样，也实现了 Shape 接口，如代码清单 22 所示。

代码清单 22：Triangle 类的源代码

```
public class Triangle implements Shape
{
    public void draw()
    {
```

```
            System.out.println("Triangle.draw()");
        }
        public void erase()
        {
            System.out.println("Triangle.erase()");
        }
    }
```

为了提供出错管理，这里特别提供一个 BadShapeException 异常类。如果客户端请求一个并不存在的 Shape 的话，工厂就应当抛出这个异常，如代码清单 23 所示。

代码清单 23：BadShapeException 类的源代码

```
public class BadShapeException extends Exception
{
    public BadShapeException(String msg)
    {
        super(msg);
    }
}
```

4．描图员（ArtTracer）系统的使用方法如代码清单 24 所示。

代码清单 24：描图员（ArtTracer）系统的代码

```
try
{
    ArtTracer art = new ArtTracer();
    art.factory("circle");
    art.factory("square");
    art.factory("triangle");
    art.factory("diamond");
}
catch(BadShapeException e)
{
    ...
}
```



对 ArtTracer 类提出菱形（diamond）请求时，会收到 BadShapeException 异常。

参考文献

[SINTES00] Tony Sintes. Polymorphism in its Purest Form–The Nature of Abstract Classes and Polymorphism. JavaWorld (www.javaworld.com)，December 2000

第 13 章　工厂方法（Factory Method）模式

工厂方法模式是类的创建模式，又叫做虚拟构造子（Virtual Constructor）模式或者多态性工厂（Polymorphic Factory）模式。

工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。

13.1　引　　言

在阅读本章之前，请首先阅读本书的“简单工厂（Simple Factory）模式”一章。

简单工厂模式的优缺点

正如本书在“简单工厂（Simple Factory）模式”一章里介绍过的，工厂模式有简单工厂模式、工厂方法模式和抽象工厂模式几种。

在简单工厂模式中，一个工厂类处于对产品类实例化的中心位置上，它知道每一个产品，它决定哪一个产品类应当被实例化。这个模式的优点是允许客户端相对独立于产品创建的过程，并且在系统引入新产品的时候无需修改客户端，也就是说，它在某种程度上支持“开–闭”原则。

这个模式的缺点是对“开–闭”原则的支持不够，因为如果有新的产品加入到系统中去，就需要修改工厂类，将必要的逻辑加入到工厂类中。

工厂方法模式的引进

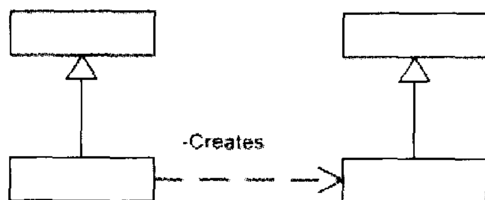
本章要讨论的工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。

首先，在工厂方法模式中，核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给子类去做。这个核心类则摇身一变，成为了一个抽象工厂角色，仅负责给出具体工厂子类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

这种进一步抽象化的结果，使这种工厂方法模式可以用来允许系统在不修改具体工厂角色的情况下引进新的产品，这一特点无疑使得工厂模式具有超过简单工厂模式的优越

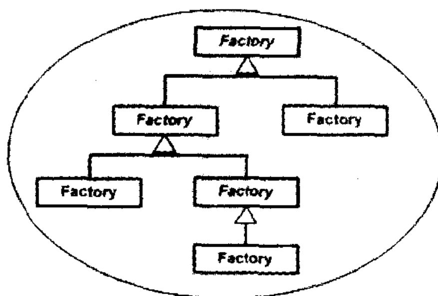
性。

下图所示是工厂方法模式的简略类图，这个类图中仅显示了一个工厂类和一个产品类，而在实际系统里面，会遇到多个产品类以及相应的工厂类。

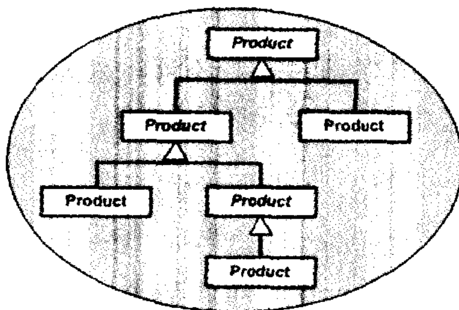


平行的等级结构

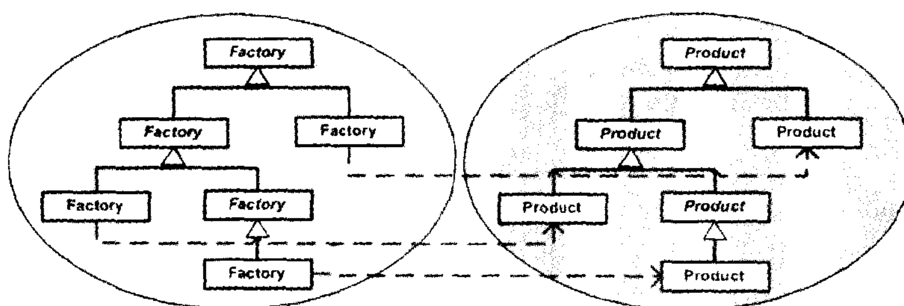
在一个系统设计中，常常是首先有产品角色，然后有工厂角色。在可以应用工厂方法模式的情形下，一般都会有一个产品的等级结构，由一个（甚至多个）抽象产品和多个具体产品组成。产品的等级结构如下图所示，树图中有阴影的是树枝型节点。



在上面的产品等级结构中，出现了多于一个的抽象产品类，以及多于两个的层次。这其实是真实的系统中常常出现的情况。当将工厂方法模式应用到这个系统中去的时候，常常采用的一个做法是按照产品的等级结构设计一个同结构的工厂等级结构。工厂的等级结构如下图所示，树图中有阴影的是树枝型节点。



然后由相应的工厂角色创建相应的产品角色，工厂方法模式的应用如下图所示，图中的虚线代表创建（依赖）关系。

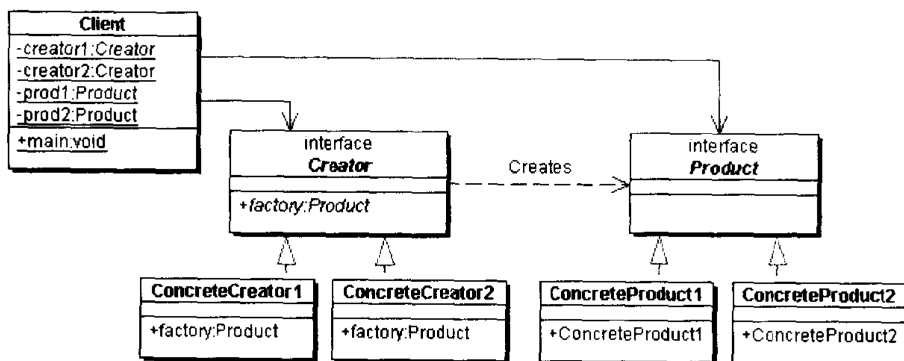


工厂方法模式并没有限制产品等级结构的层数。一般的书籍中都以两个层次为例，第一层是抽象产品层，第二层是具体产品层。但是在实际的系统中，产品常常有更为复杂的层次。

13.2　工厂方法模式的结构

结构与角色

为了说明工厂方法模式的结构，下面以一个最简单的情形为例。这个示意性系统的类图如下图所示。



从上图可以看出，这个使用了工厂方法模式的系统涉及到以下的角色：

- 抽象工厂（Creator）角色：担任这个角色的是工厂方法模式的核心，它是与应用程序无关的。任何在模式中创建对象的工厂类必须实现这个接口。在上面的系统中，这个角色由 Java 接口 Creator 扮演；在实际的系统中，这个角色也常常使用抽象 Java 类实现。
- 具体工厂（Concrete Creator）角色：担任这个角色的是实现了抽象工厂接口的具体 Java 类。具体工厂角色含有与应用密切相关的逻辑，并且受到应用程序的调用以创建产品对象。在本系统中给出了两个这样的角色，也就是具体 Java 类 ConcreteCreator1 和 ConcreteCreator2。
- 抽象产品（Product）角色：工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。在本系统中，这个角色由 Java 接口 Product

扮演；在实际的系统中，这个角色也常常使用抽象 Java 类实现。

- 具体产品（Concrete Product）角色：这个角色实现了抽象产品角色所声明的接口。工厂方法模式所创建的每一个对象都是某个具体产品角色的实例。

在本系统中，这个角色由具体 Java 类 ConcreteProduct1 和 ConcreteProduct2 扮演，它们都实现了 Java 接口 Product。

最后，为了说明这个系统的使用办法，特地引进了一个客户端角色 Client。这个角色创建工厂对象，然后调用工厂对象的工厂方法创建相应的产品对象。

源代码

下面就是这个示意性系统的源代码。

首先是抽象工厂角色的源代码，如代码清单 1 所示。这个角色是用一个 Java 接口实现的，它声明了一个工厂方法，要求所有的具体工厂角色都实现这个工厂方法。

代码清单 1：抽象工厂角色 Creator 类的源代码

```
package com.javapatterns.factorymethod;
public interface Creator
{
    /**
     *  工厂方法
     */
    public Product factory();
}
```

下面是抽象产品角色的源代码，如代码清单 2 所示。由于这里考虑的是最为简单的情形，所以抽象产品角色仅仅为具体产品角色提供一个共同的类型而已，所以是用一个 Java 标识接口实现的。一个没有声明任何方法的接口叫做标识接口，关于标识接口的讨论请见本书的“专题：Java 接口”一章。

代码清单 2：抽象产品角色 Product 类的源代码

```
package com.javapatterns.factorymethod;
public interface Product
{
}
```

下面是具体工厂角色 ConcreteCreator1 的源代码，如代码清单 3 所示。这个角色实现了抽象工厂角色 Creator 所声明的工厂方法。

代码清单 3：具体工厂角色 ConcreteCreator1 类的源代码

```
package com.javapatterns.factorymethod;
public class ConcreteCreator1 implements Creator
{
    /**
     *  工厂方法
```



```
    */
    public Product factory()
    {
        return new ConcreteProduct1();
    }
}
```

下面是具体工厂角色 ConcreteCreator2 的源代码，如代码清单 4 所示。与 Concrete Creator1 一样，这个角色实现了抽象工厂角色 Creator 所声明的工厂方法。

代码清单 4：具体工厂角色 ConcreteCreator2 类的源代码

```
package com.javapatterns.factorymethod;
public class ConcreteCreator2 implements Creator
{
    /**
     * 工厂方法
     */
    public Product factory()
    {
        return new ConcreteProduct2();
    }
}
```

下面是具体产品角色 CocnreteProduct1 的源代码，如代码清单 5 所示。它是此系统向客户端提供的产品，在通常情况下，这个类会有复杂的商业逻辑。

代码清单 5：具体产品角色 ConcreteProduct1 类的源代码

```
package com.javapatterns.factorymethod;
public class ConcreteProduct1 implements Product
{
    public ConcreteProduct1()
    {
        //do something
    }
}
```

类似地，下面是具体产品角色 ConcreteProduct2 的源代码，如代码清单 6 所示。

代码清单 6：具体产品角色 ConcreteProduct2 类的源代码

```
package com.javapatterns.factorymethod;
public class ConcreteProduct2 implements Product
{
    public ConcreteProduct2()
    {
        //do something
    }
}
```

这个示意性的系统只给出了两个具体工厂类和两个具体产品类。在真实的系统中，工厂方法模式一般都会涉及到更多的具体工厂类和更多的具体产品类。

下面就是客户端角色的源代码，如代码清单 7 所示。

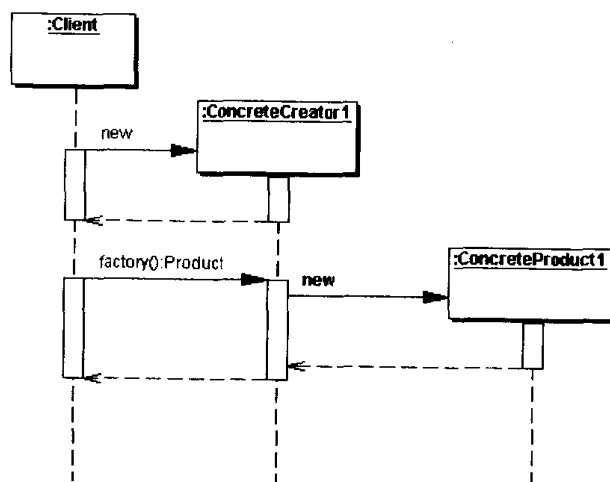
代码清单 7：客户端角色 Client 类的源代码

```
package com.javapatterns.factorymethod;
public class Client
{
    private static Creator creator1, creator2;
    private static Product prod1, prod2;
    public static void main(String[] args)
    {
        creator1 = new ConcreteCreator1();
        prod1 = creator1.factory();
        creator2 = new ConcreteCreator2();
        prod2 = creator2.factory();
    }
}
```

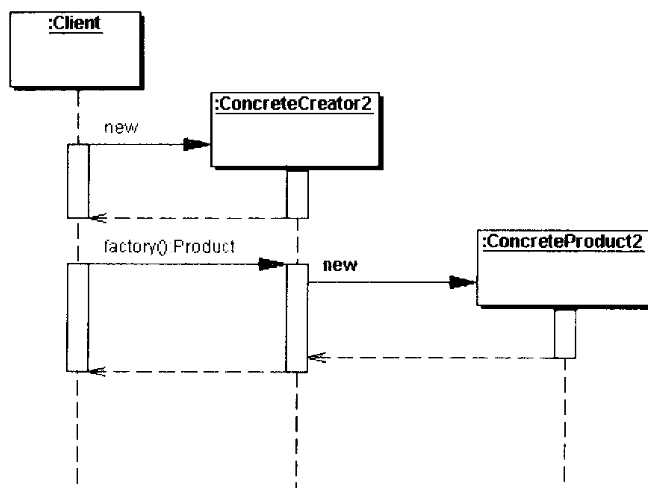
工厂方法模式的活动序列图

Client 对象的活动可以分成两部分。

- 客户端创建 ConcreteCreator1 对象。这时客户端所持有变量的静态类型是 Creator，而实际类型是 ConcreteCreator1。然后，客户端调用 ConcreteCreator1 对象的工厂方法 factory()，接着后者调用 ConcreteProduct1 的构造子创建出产品对象。其时序图如下图所示。



- 客户端创建一个 ConcreteCreator1 对象，然后调用 ConcreteCreator2 对象的工厂方法 factory()，而后者调用 ConcreteProduct2 的构造子创建出产品对象，其时序图如下图所示。



工厂方法模式和简单工厂模式

工厂方法模式和简单工厂模式在结构上的不同是很明显的。工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上。工厂方法模式可以允许很多具体工厂类从抽象工厂类中将创建行为继承下来，从而可以成为多个简单工厂模式的综合，进而推广了简单工厂模式。

工厂方法模式退化后可以变得很像简单工厂模式。设想如果非常确定一个系统只需要一个具体工厂类，那么就不妨把抽象工厂类合并到具体的工厂类中去。由于只有一个具体工厂类，所以不妨将工厂方法改为静态方法，这时候就得到了简单工厂模式。

与简单工厂模式中的情形一样的是，ConcreteCreator 的 factory() 方法返还的数据类型是一个抽象类型 Product，而不是哪一个具体产品类型，而客户端也不必知道所得到的产品的真实类型。这种多态性设计将工厂类选择创建哪一个产品对象、如何创建这个对象的细节完全封装在具体工厂类内部。

工厂方法模式之所以有一个别名叫多态性工厂模式，显然是因为具体工厂类都有共同的接口，或者都有共同的抽象父类。

如果系统需要加入一个新的产品，那么所需要的就是向系统中加入一个这个产品类以及它所对应的工厂类。没有必要修改客户端，也没有必要修改抽象工厂角色或者其他已有的具体工厂角色。对于增加新的产品类而言，这个系统完全支持“开–闭”原则。

13.3　工厂方法模式在农场系统中的实现

系统的优化

现在继续考察农场的管理系统。在本书的“简单工厂（Simple Factory）模式”一章里，讨论了支持水果类作物的系统。在那个系统中，有一个全知全能的园丁角色

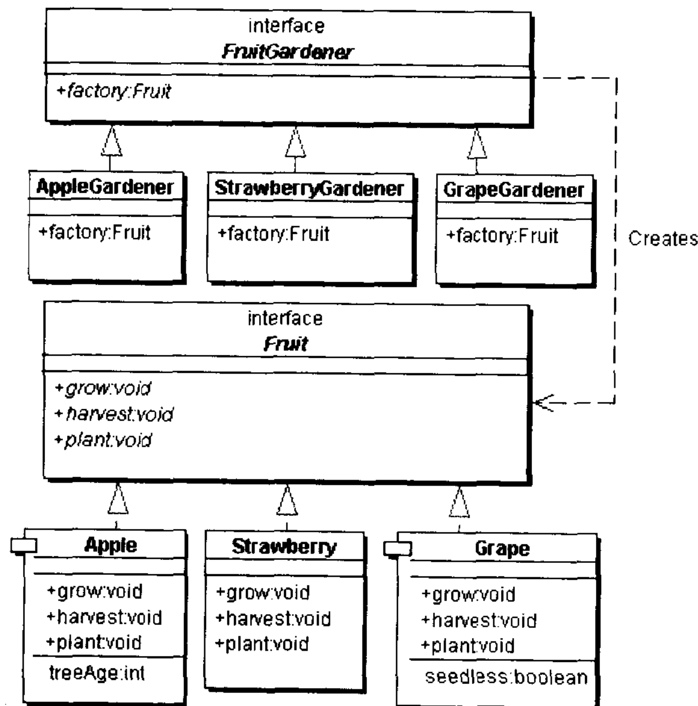
（FruitGardener），控制所有作物的种植、生长和收获。现在这个农场的规模变大了，而同时发生的是管理更加专业化了。过去的全能人物没有了，每一种农作物都有专门的园丁管理，形成规模化和专业化生产。

系统的设计

取代了过去的全能角色的是一个抽象的园丁角色，这个角色规定出具体园丁角色需要实现的具体职能，而真正负责作物管理的则是负责各种作物的具体园丁角色。

这一章仍然考虑前面所讨论过的植物，包括葡萄（Grape）、草莓（Strawberry）以及苹果（Apple）等。专业化的管理要求有专门的园丁负责专门的水果，比如苹果由“苹果园丁”负责，草莓有“草莓园丁”负责，而葡萄由“葡萄园丁”负责。这些“苹果园丁”、“草莓园丁”以及“葡萄园丁”都是实现了抽象的“水果园丁”接口的具体工厂类，而“水果园丁”则扮演抽象工厂角色。

这样一来，农场系统的设计图就如下图所示。

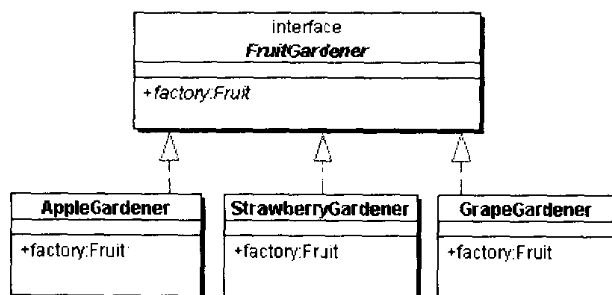


抽象工厂类 FruitGardener 是工厂方法模式的核心，但是它并不掌握水果类或蔬菜类的生杀大权。相反地，这项权力被交给子类，即 AppleGardener StawberryGardener 以及 GrapeGardener。

工厂角色的等级结构

各个工厂角色组成一个工厂的等级结构，如下图所示。





在上图所示的等级结构中，FruitGardener 是所有具体工厂角色的超类。在本系统中，这个抽象角色是由 Java 接口 FruitGardener 实现的，它声明了一个工厂方法，要求所有的具体工厂角色都实现这个工厂方法。

这个角色的源代码如代码清单 8 所示。

代码清单 8：抽象工厂角色 FruitGardener 的源代码

```
package com.javapatterns.factorymethod.farm;
public interface FruitGardener
{
    /**
     *  工厂方法
     */
     public Fruit factory();
}
```

AppleGardener 类是具体工厂类，它实现了 FruitGardener 接口，提供了工厂方法的实现。它的源代码如代码清单 9 所示。

代码清单 9：具体工厂类 AppleGardener 的源代码

```
package com.javapatterns.factorymethod.farm;
public class AppleGardener
     implements FruitGardener
{
    /**
     *  工厂方法
     */
     public Fruit factory()
     {
          return new Apple();
     }
}
```

StrawberryGardener 类是一个具体工厂类，与 AppleGardener 一样，它也实现了 FruitGardener 接口。此类的源代码如代码清单 10 所示。

代码清单 10：具体工厂类 StrawberryGardener 的源代码

```
package com.javapatterns.factorymethod.farm;
```

```
public class StrawberryGardener
      implements FruitGardener
{
   /**
    *  工厂方法
    */
    public Fruit factory()
    {
         return new Apple();
    }
    }
```

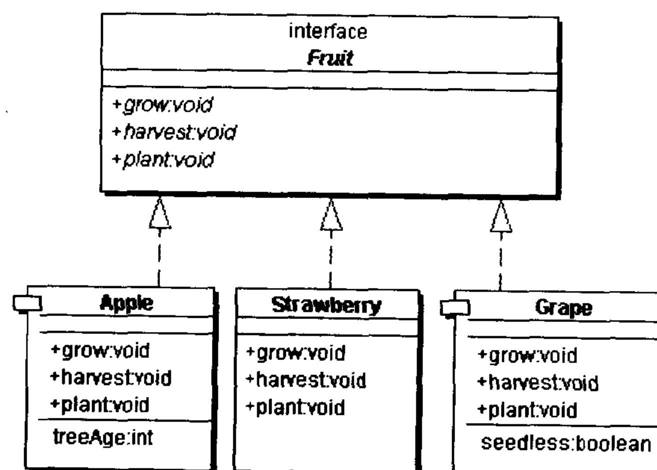
同样，具体类 GrapeGardener 的源代码如代码清单 11 所示。

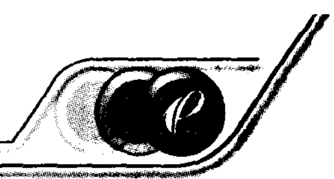
代码清单 11：具体工厂类 GrapeGardener 的源代码

```
package com.javapatterns.factorymethod.farm;
public class GrapeGardener
      implements FruitGardener
{
   /**
    *  工厂方法
    */
    public Fruit factory()
    {
         return new Apple();
    }
}
```

产品角色的等级结构

所有的产品角色都属于一个产品的等级结构，在这个等级结构最上面的是抽象产品角色 Product。在本系统中，这个角色是由一个 Java 接口 Fruit 实现的。由各个工厂角色组成的工厂等级结构如下图所示。





所有具体产品都必须实现抽象产品 Fruit 所声明的接口，如代码清单 12 所示。

代码清单 12：抽象产品角色 Fruit 的源代码

```
package com.javapatterns.factorymethod.farm;
public interface Fruit
{
    void grow();
    void harvest();
    void plant();
}
```

Apple 类是一个具体产品类，实现了 Fruit 接口，如代码清单 13 所示。

代码清单 13：水果类 Apple 的源代码

```
package com.javapatterns.factorymethod.farm;
public class Apple implements Fruit
{
    private int treeAge;
    public void grow()
    {
        System.out.println("Apple is growing...");
    }
    public void harvest()
    {
        System.out.println("Apple has been harvested.");
    }
    public void plant()
    {
        System.out.println("Apple has been planted.");
    }
    public int getTreeAge()
    {
        return treeAge;
    }
    public void setTreeAge(int treeAge)
    {
        this.treeAge = treeAge;
    }
}
```

具体产品类 Strawberry 的源代码如代码清单 14 所示。

代码清单 14：水果类 Strawberry 的源代码

```
package com.javapatterns.factorymethod.farm;
public class Strawberry implements Fruit
{
        public void grow()
```

```
    {
        System.out.println("Strawberry is growing...");
    }
    public void harvest()
    {
        System.out.println("Strawberry has been harvested.");
    }
    public void plant()
    {
        System.out.println("Strawberry has been planted.");
    }
}
```

Grape 类也是一个具体产品角色，如代码清单 15 所示。

代码清单 15：水果类 Grape 的源代码

```
package com.javapatterns.factorymethod.farm;
public class Grape implements Fruit
{
    private boolean seedless;
    public void grow()
    {
        System.out.println("Grape is growing...");
    }
    public void harvest()
    {
        System.out.println("Grape has been harvested.");
    }
    public void plant()
    {
        System.out.println("Grape has been planted.");
    }
    public boolean getSeedless()
    {
        return seedless;
    }
    public void setSeedless(boolean seedless)
    {
        this.seedless = seedless;
    }
}
```

13.4 关于工厂方法模式的实现

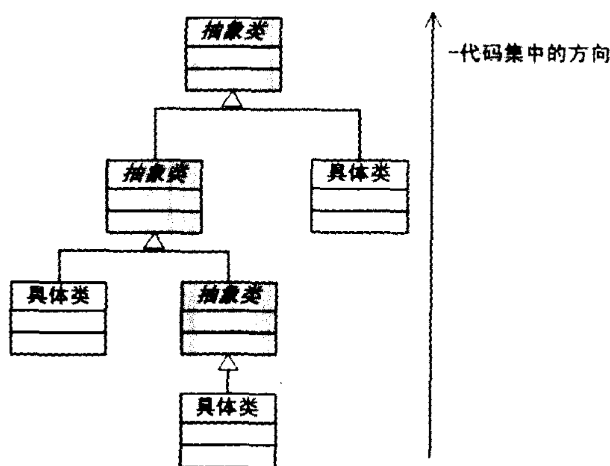
在实现工厂方法模式时，会有一些与简单工厂模式相同或者相似的地方值得讨论。



使用 Java 接口或者 Java 抽象类

抽象工厂角色和抽象产品角色都可以选择由 Java 接口或者 Java 抽象类来实现。

如果具体工厂角色具有共同的逻辑，那么这些共同的逻辑就可以向上移动到抽象工厂角色中，这也就意味着抽象工厂角色应当用一个 Java 抽象类实现，并由抽象工厂角色提供默认的工厂方法。相反的话，就应当用一个 Java 接口实现，对抽象产品角色也是一样。共同的逻辑应当移动到超类中去，如下图所示。



关于 Java 抽象类与 Java 接口的区别，读者可以参阅本书的“专题：抽象类”和“专题：Java 接口”两章。

使用多个工厂方法

抽象工厂角色可以规定出多于一个的工厂方法，从而使具体工厂角色实现这些不同的工厂方法。这些方法可以提供不同的商业逻辑，以满足提供不同的产品对象的任务。

最后，在给相关的类和方法取名字时，应当注意让别人一看便知道在系统的设计中使用了工厂模式。

产品的循环使用

读者可以参阅本书在“工厂方法（Factory Method）模式”一章中做过的类似的分析。

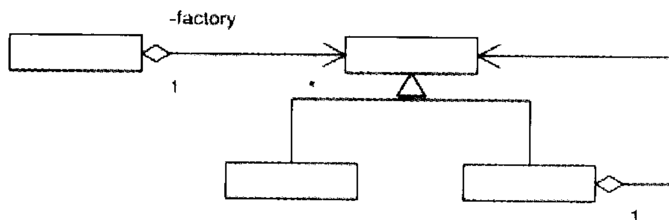
在前面给出的示意性系统中，工厂方法总是调用产品类的构造子以创建一个新的产品实例，然后将这个实例提供给客户端。而在实际情形中，工厂方法所做的事情可以相当复杂。

一个常见的复杂工厂逻辑就是循环使用产品对象。如果产品对象可以由内部状态表征的话，那么对于每一个可能的内部状态，往往仅需要一个产品实例。

这时候，工厂对象就需要将已经创建过的产品对象登记到一个聚集里面，然后根据客

户端所请求的产品状态，向聚集进行查询。如果聚集中有这样的产品对象，那么就直接将这个产品对象返还给客户端；如果聚集中没有这样的产品对象，那么就创建一个新的满足要求的产品对象，然后将这个对象登记到聚集中，再返还给客户端。

享元模式就使用了这样的循环工厂模式，如下图所示。



关于享元模式，请读者参阅本书的“享元模式（Flyweight Pattern）”一章。

多态性的丧失和模式的退化

一个工厂方法模式的实现依赖于工厂角色和产品角色的多态性。在有些情况下，这个模式可以出现退化，其特征就是多态性的丧失。

工厂方法创建对象

正如前面所讨论的，工厂方法不一定每一次都返还一个新的对象。但是它所返还的对象一定是他自己创建的，而不是在一个外部对象里面创建，然后传入到工厂对象中的。

但是，这是否意味着凡是返还一个新的对象的方法都是工厂方法呢？不一定。

工厂方法返还的类型

工厂方法返还的应当是抽象类型，而不是具体类型，只有这样才能保证针对产品的多态性。换言之，调用工厂方法的客户端可以针对抽象编程，依赖于一个抽象产品类型，而不是具体产品类型。

在特殊情况下，工厂方法仅仅返还一个具体产品类型。这个时候工厂方法模式的功能就退化了，表现为针对产品角色的多态性的丧失。换言之，客户端从工厂方法的静态类型可以得知将要得到的是什么类型的对象，而这违背了工厂方法模式的用意。

当工厂方法模式发生上面的退化时，就不再是工厂方法模式了[METSKER02]。在任何一个面向对象的语言里面，都会有大量的方法返还一个新的对象，很多设计师将这种方法都叫做“工厂方法”，但是并不是所有的这种工厂方法都是工厂方法模式。本章后面的问答题给出了两个例子，读者可以参考。

工厂等级结构

工厂对象应当有一个抽象的超类型。换言之，应当有数个具体工厂类作为一个抽象超类型的具体子类存在于工厂等级结构中。如果等级结构中只有一个具体工厂类的话，那么抽象工厂角色也可以省略。

当抽象工厂角色被省略时，工厂方法模式就发生了退化，这一退化表现为针对工厂角色的多态性的丧失。这种工厂方法模式仍然可以发挥一部分工厂方法模式的用意，因此叫

做退化的工厂方法模式。

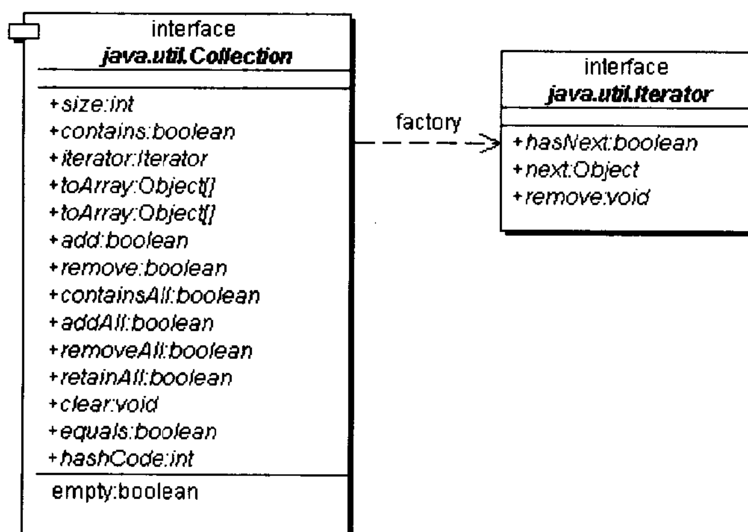
此时经常可以由简单工厂模式代替。

13.5 Java 语言中工厂方法模式的例子

工厂方法模式是一个很常见的设计模式，可以在 Java 语言 API 的各个角落里面找到。下面就给出一些读者可能已经熟悉的例子，并从模式的角度上加以分析。

在 Java 聚集中的应用（之一）

Java 聚集是 Java 1.2 版提出来的。多个对象聚在一起形成的总体称之为聚集（Aggregate），聚集对象是能够包容一组对象的容器对象。所有的 Java 聚集都实现 java.util.Collection 接口，这个接口规定所有的 Java 聚集必须提供一个 iterator()方法，返还一个 Iterator 类型的对象，如下图所示。



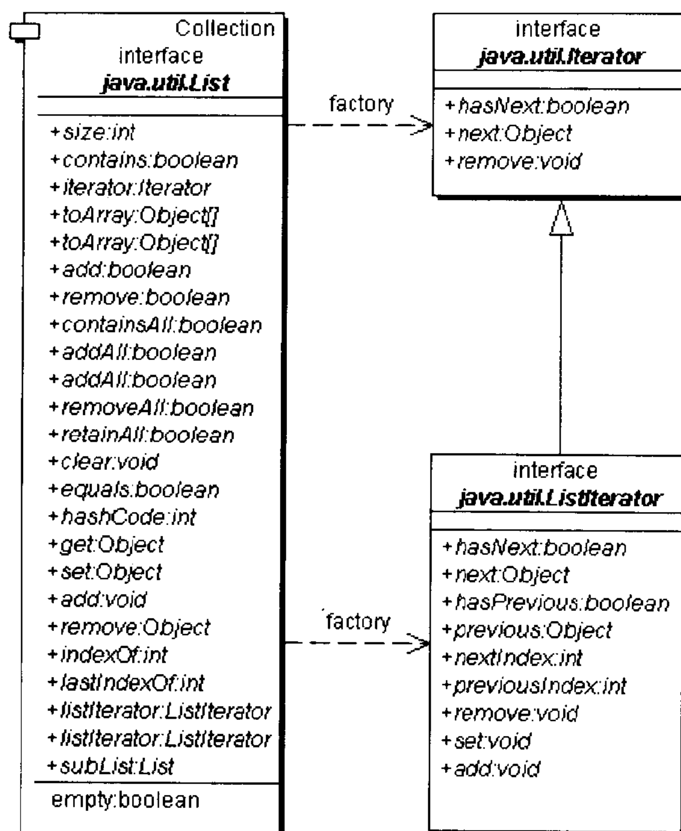
一个具体的 Java 聚集对象会通过这个 iterator()方法接口返还一个具体的 Iterator 类。可以看出，这个 iterator()方法就是一个工厂方法。

这个 Iterator 接口是迭代子模式的应用。关于迭代子模式，请参阅本书的“迭代子（Iterator）模式”一章。

在 Java 聚集中的应用（之二）

类似地，Java 的列是一种特殊的 Java 聚集。所有的 Java 除了实现 Collection 接口之外，还实现了一个 List 接口。这个 List 接口给出了两个工厂方法，一个是上面已经讨论过的 iterator()方法，返还一个 Iterator 类型的对象；另一个是 listIterator()方法，返还一个 ListIterator 类型的对象。

有趣的是，Iterator 和 ListIterator 组成一个继承的等级结构，后者是前者的子接口。如下图所示。



这个 Iterator 接口和 ListIterator 均是迭代子模式的应用。关于迭代子模式以及 Java 语言 API 对迭代子模式的支持，请参见本书的“迭代子（Iterator）模式”以及“专题：Java 对迭代子模式的支持”等章。

URL 与 URLConnection 的应用

URL 类代表一个 Uniform Resource Locator，也就是互联网上资源的一个指针。而一个资源可以是一个简单的文件或者目录，也可以是一个更加复杂的对象，比如向数据库或者搜索引擎查询对象，http://www.yahoo.com 就是一个合法的 URL。

创建一个 URL 对象也很简单，只要将一个合法的 URL 传入到 URL 的构造子中即可，如代码清单 16 所示。

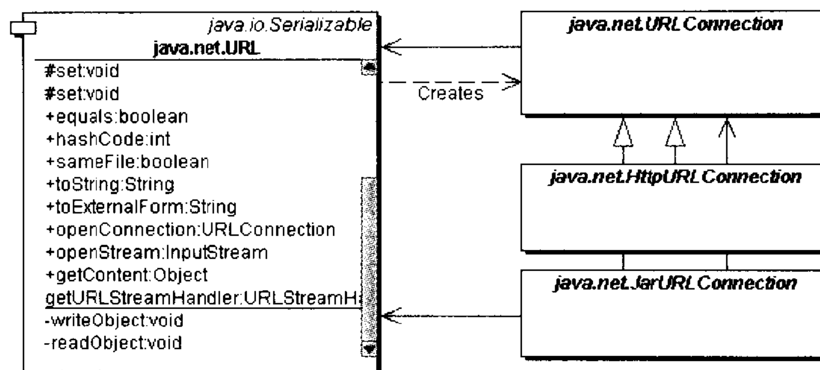
代码清单 16：怎样将 URL 类实例化

```
URL url = new URL("http://www.yahoo.com");
```

URL 对象提供一个叫做 openConnection()的工厂方法，这个方法返还一个 URL Connection 对象。而 URLConnection 对象则代表一个与远程对象的连接。URLConnection 是所有的代表应用系统与一个 URL 的连接对象的超类，使用 URLConnection 对象可以针对一个 URL 进行读写操作。



显然，URL 对象使用了工厂方法模式，以一个工厂方法 openConnection()返还一个 URLConnection 类型的对象。由于 URLConnection 是一个抽象类，因此所返还的不可能是这个抽象类的实例，而必然是其具体子类的实例。URL 类和 URLConnection 类及其子类的结构图如下图所示。



为了说明 URL 作为一个工厂类，而 URLConnection 作为一个抽象产品类的使用，下面提供一个简单的例子，如代码清单 17 所示。

代码清单 17：一个例子的源代码

```
package com.javapatterns.factorymethod.url;
import java.net.*;
import java.io.*;
public class URLConnectionReader
{
    public static void main(String[] args) throws Exception
    {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
            yc.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

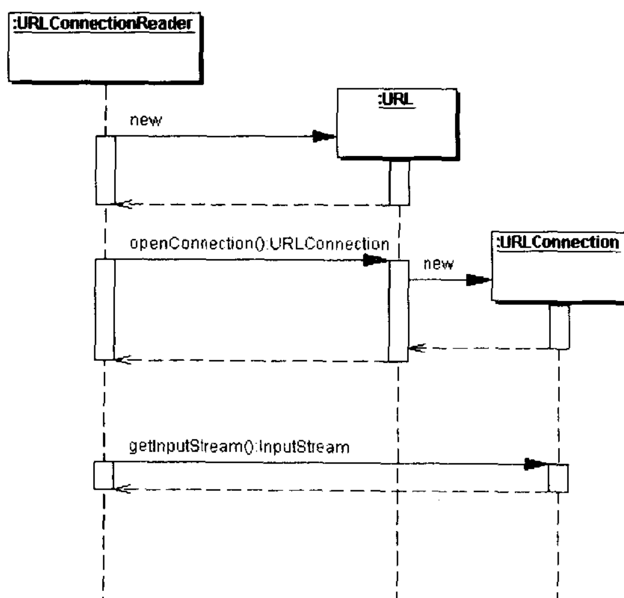
可以看出，这个例子在运行时的活动顺序如下：

（1）创建一个以 http://www.yahoo.com 为目标的 URL 对象。

（2）调用 URL 对象的 openConnection()方法，得到一个 http://www.yahoo.com 的远程连接对象，这个对象的类型是 URLConnection。

（3）客户端调用 URLConnection 对象的 getInputStream()方法读入远程 URL 的数据。

系统的时序图如下图所示。

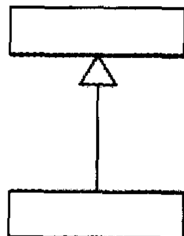


这个系统在运行时会打印出 http://www.yahoo.com 主页的全部 HTML 源代码。

13.6 工厂方法模式与其他模式的关系

模版方法模式

工厂方法模式常常与模版方法模式一起联合使用。读者可能会问，这两种模式一个是关于对象的创建，另一个是关于对象的行为，为什么会常常一起使用呢？



原因其实不难理解：第一，两个模式都是基于方法的，工厂方法模式是基于多态性的工厂方法的，而模版方法模式是基于模版方法和基本方法的；第二，两个模式都将具体工作交给子类。工厂方法模式将创建工作推延给子类，模版方法模式将剩余逻辑交给子类。模版方法模式的简略类图如右图所示。

从各个工厂角色组成的工厂等级结构上看，抽象工厂角色中可以加入一个模版方法，代表某个顶级逻辑。而这个模版方法调用几个基本方法，这些基本方法中就可以有一个或者多个工厂方法。这些工厂方法代表剩余逻辑，交给具体子类实现。

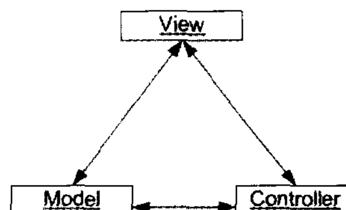
模版方法本身也可能是工厂方法，它的对象创建过程就是所谓的顶级结构。而这个过程可以分为数个具体步骤，每一个步骤都是顶级逻辑的组成部分。

换言之，使用模版方法模式可以将某一个顶级行为分解成为数个创建行为，由子类中的工厂方法体现出来。不同的具体工厂类则提供了顶级逻辑中的剩余逻辑的不同实现。通过使用不同的具体工厂对象，客户端可以达到使用不同版本的顶级逻辑的目的。

本章在问答题中给出一个例子，详细地给出了有关模版方法模式和工厂方法模式的讨论。关于模版方法模式的介绍，请参见本书的“模版方法（Template Method）模式”一章。

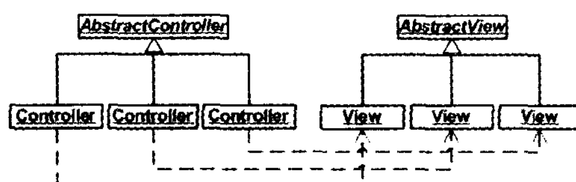
MVC 模式

MVC 模式并不是严格意义上的设计模式，而是在更高层次上的架构模式。MVC 模式可以分解成为几个设计模式的组合，包括合成模式、策略模式、观察者模式，也有可能会包括装饰模式、调停者模式、迭代子模式以及工厂方法模式等。



关于 MVC 模式的讨论可以参考本书的“专题：MVC 模式与用户输入数据检查”一章。MVC 模式的结构图如右图所示。

工厂方法模式总是涉及到两个等级结构中的对象，而这两个等级结构可以分别是 MVC 模式中的控制器（Controller）和视图（View）。一个 MVC 模式可以有多个控制器和多个视图，如下图所示。

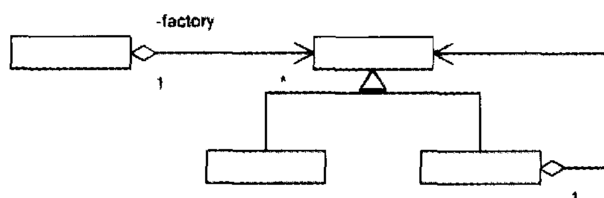


换言之，控制器端可以创建合适的视图端，就如同工厂角色创建合适的对象角色一样，模型端则可以充当这个创建过程的客户端。

如果系统内只需要一个控制器，那么设计可能简化为简单工厂模式。请参见本书的“简单工厂（Simple Factory）模式”一章。

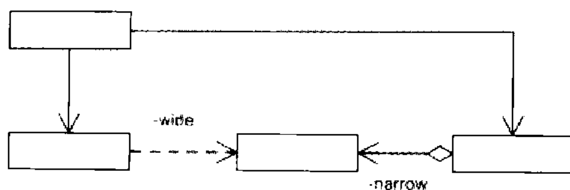
享元模式

正如在本章“关于工厂方法模式的实现”一节中讨论过的，享元模式使用了带有循环逻辑的工厂方法。享元模式的简略类图如下图所示。



备忘录模式

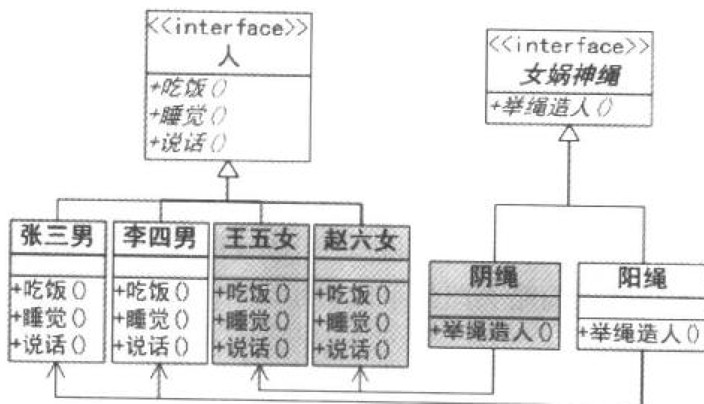
享元模式使用了一个聚集来登记所创建的产品对象，以便可以通过查询这个聚集找到和共享已经创建了的产品对象，这就是备忘录模式的应用。备忘录模式的简略类图如下图所示。



13.7　女娲举绳造人

仍以女娲造人的故事为例，《风俗通》中说："俗说天地开辟，未有人民。女娲抟黄土为人，剧务，力不暇供，乃引绳于桓泥中，举以为人。"参考"简单工厂模式"一章，女娲造人的初期使用的是简单工厂模式。然后，女娲发现她不能用这种方法造出所有的人，于是她就想出了一个聪明的办法：使用一根绳子，在泥水里搅，然后一甩，所有的泥点都变成了人。当然，女娲造出的人有男女之别，是因为女娲使用的绳子有阴阳之分。在下面，读者可以看出，女娲采用了工厂方法模式来达到造人的目的。

女娲举绳造人的故事是应用工厂方法模式的一个实例。抽象角色"女娲神绳"是系统的中心，但是它仅仅声明出"举绳造人"方法，而没有实现它。真正做这个工作的是具体工厂类，也就是"阴绳"和"阳绳"类。工厂方法模式在女娲举绳造人系统中的实现如下图所示。



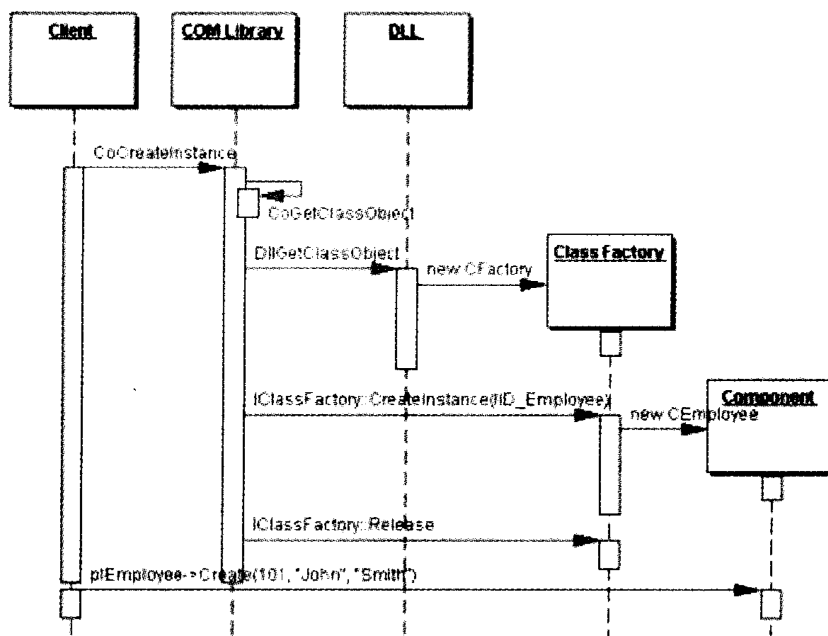
13.8　其他的例子

COM 技术架构中的工厂方法模式

在微软公司所提倡的 COM（Component Object Model）技术架构中，工厂方法模式起着关键的作用。

在 COM 架框里，Creator 接口的角色是由一个叫做 IClassFactory 的 COM 接口来担任

的。而具体类 ConcreteCreator 的角色是由实现 IClassFactory 接口的类 CFactory 来担任的。一般而言，对象的创建可能要求分配系统资源，要求在不同的对象之间进行协调等。因为 IClassFactory 的引进，所有这些在对象的创建过程中出现的细节问题，都可以封装在一个实现 IClassFactory 接口的具体的工厂类里面。这样一来，一个 COM 架构的支持系统只需要创建这个工厂类 CFactory 的实例就可以了。微软的 COM（Component Object Model）技术架构如下图所示。



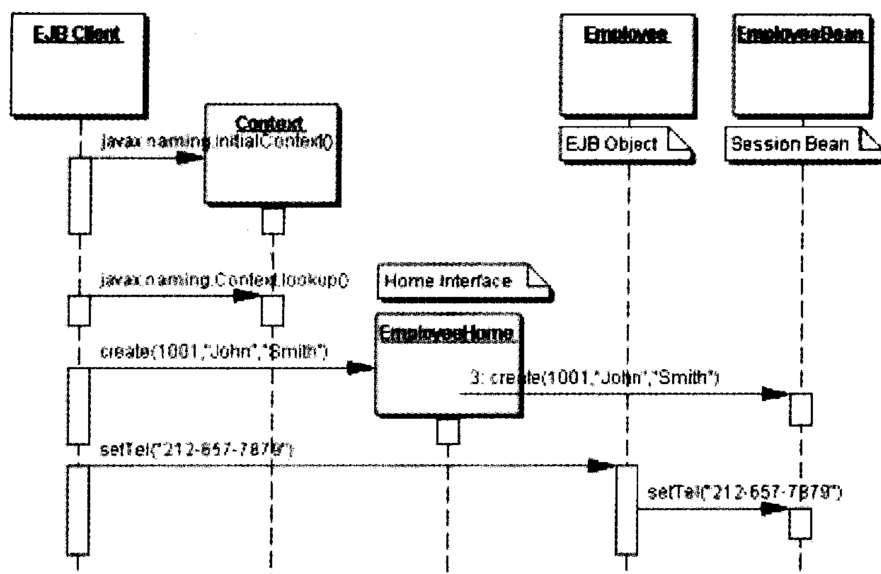
在上面的序列活动（Sequence Activity）图中，用户端调用 COM 的库函数 CoCreateInstance。CoCreateInstance 在 COM 架构中以 CoGetClassObject 实现。CoCreateInstance 会在视窗系统的 Registry 里搜寻所要的构件（在这个例子中即 Cemployee）。如果找到了这个构件，就会加载支持此构件的 DLL。当此 DLL 加载成功后，CoGetClassObject 就会调用 DllGetClassObject。后者使用 new 操作符将工厂类 CFactory 实例化。

下面，DllGetClassObject 会向工厂类 Cfactory 查询 IClassFactory 接口，返还给 CoCreate Instance。CoCreateInstance 接下来利用 IClassFactory 接口调用 CreateInstance 函数。此时，IClassFactory::CreateInstance 调用 new 操作符来创建所要的构件（Cemployee）。此外，它搜寻 IEmployee 接口。在拿到接口的指针后，CoCreateInstance 释放掉工厂类并把接口的指针返还给客户端。

客户端现在就可以利用这个接口调用此构件中的方法了。

EJB 技术架构中的工厂方法模式

Sun Microsystem 所倡导的 EJB（Enterprise Java Beans）技术架构是一套为 Java 语言设计的、用来开发企业规模应用系统的构件模型。为了说明 EJB 架构是怎样利用工厂方法模式的，请考察下图所示的序列活动图。



在 EJB 技术架构中，工厂方法模式也起着关键的作用。

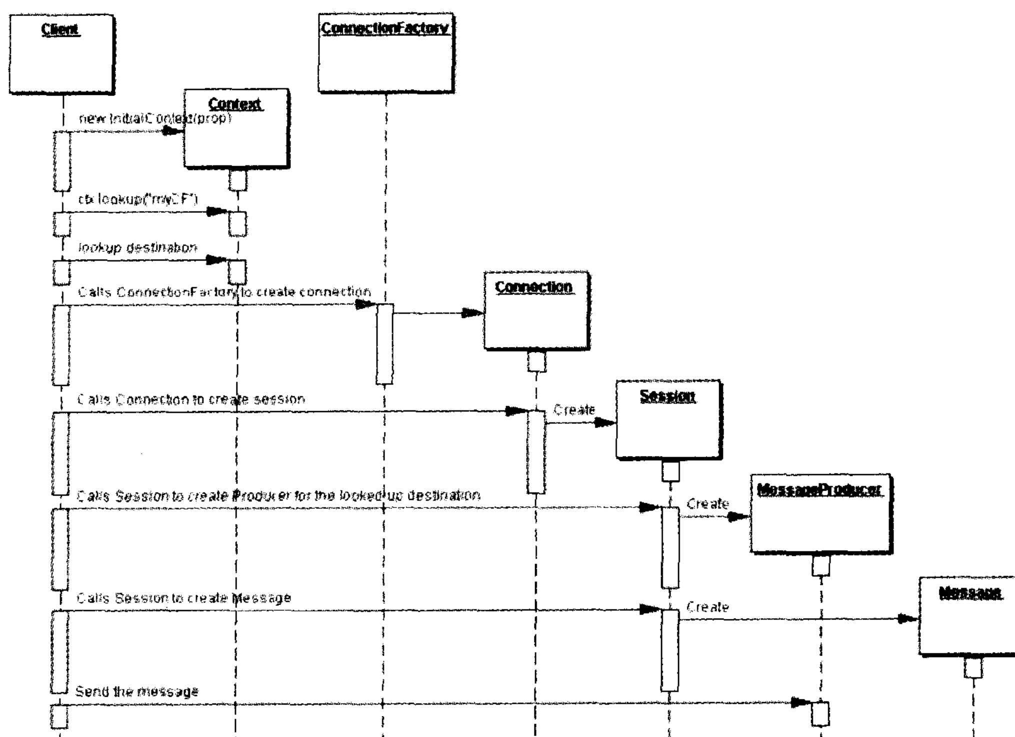
在这个时序图中，用户端首先创建一个新的 Context 对象，以便利用 JNDI 服务器寻找 EJBObject。在得到这个 Context 对象后，就可以使用 JNDI 名，比如 Employee，来拿到 EJB 类 Employee 的 Home 接口。通过使用 Employee 的 Home 接口，客户端可以创建 EJB 对象，比如 EJB 类 Employee 的实例，然后调用 Employee 的方法，如代码清单 18 所示。

代码清单 18： Home 接口提供工厂方法的实例

```
// 取到 JNDI naming context
Context ctx = new InitialContext ();
// 利用 ctx 索取 EJB Home 接口
EmployeeHome home = (EmployeeHome)ctx.lookup("Employee");
// 利用 Home 接口创建一个 Session Bean 对象
// 这里使用的是标准的工厂方法模式
Employee emp = home.create (1001, "John", "Smith");
// 调用方法
emp.setTel ("212-657-7879");
```

JMS 技术架构中的工厂方法模式

JMS（Java Messaging Service）定义了一套标准的 API，让 Java 语言程序能通过支持 JMS 标准的 MOM（Message Oriented Middleware 或者面向消息的中间服务器）来创建和交换消息（message）。现在来举例看一看 JMS（Java Messaging Service）技术架构是怎样使用工厂方法模式的，如下图所示。



在上面的序列图中，用户端创建一个新的 Context 对象，以便利用 JNDI 伺服器寻找 Topic 和 ConnectionFactory 对象。在得到这个 ConnectionFactory 对象后，就可以利用 Connection 创建 Session 的实例。有了 Session 的实例后，就可以利用 Session 创建 TopicPublisher 的实例，并利用 Session 创建消息实例，如代码清单 19 所示。

代码清单 19：工厂模式被用于创建 Connection、Session、Producer 的实例

```
Properties prop = new Properties();
prop.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.sun.jndi.fscontext.RefFSContextFactory");
prop.put(Context.PROVIDER_URL,  "file:C:\temp");
// 取到 JNDI context
Context ctx = new InitialContext(prop);
// 利用 ctx 索取工厂类的实例
Topic topic = (Topic) ctx.lookup("myTopic");
TopicConnectionFactory tcf = (TopicConnectionFactory) ctx.lookup("myTCF");
// 利用工厂类创建 Connection，这是典型的工厂模式
TopicConnection tCon = tcf.createTopicConnectoin();
// 利用 Connection 创建 Session 的实例，又是工厂模式
TopicSession tSess = tCon.createTopicSession(false,
     Session.AUTO_ACKNOWLEDGE);
// 利用 Session 创建 Producer 的实例，又是工厂模式
TopicPublisher publisher = tSess.createPublisher(topic);
// 利用 Session 创建消息实例，又是工厂模式
TextMesage msg = tSess.createTextMessage("Hello from Jeff");
//发送消息
```

```
publisher.publish(msg);
```

问答题

1. 有很多 Java 语言中的 API 提供一些返还新的 Java 对象的方法。能否举出两个这样的方法的例子？请问它们是工厂方法模式吗？

2. 请问下面这句话对吗？

"一个工厂方法必须返还一个新的对象。如果返还的不是一个新的对象，就不符合工厂方法模式的描述。"

3. 请问工厂方法可不可以返还在另一个对象里实例化的一个对象？

4. 某一个商业软件产品需要支持 Sybase 和 Oracle 数据库。这个系统需要这样的一个查询运行器系统，根据客户端的需要，可以随时向 Sybase 或者 Oracle 数据库引擎发出查询。请给出这样的一个系统的示意性设计，并且请考虑在设计中使用工厂方法模式是否合适。暂时可以假定所发出的查询总是同一个 SQL 语句。

（本问题和答案受到文献[SHALLOWAY02]中的一个相似的例子的启发。本书鼓励读者阅读原著）

5. 请阅读上一题的答案，并分析上题的设计与模版方法模式有无关系。

问答题答案

1. Java 对象的 toString()方法和 clone()方法会给出一个新的 Java 对象。

一个 Java 对象的 toString()方法会给出一个 String 类型的对象；而 clone()方法会给出与原对象类型相同的对象。

它们都不是工厂方法模式，因为它们都不能返还一个抽象类型，客户端事先都知道将要得到的对象类型。

换言之，并非每一个返还一个新的对象的方法都是工厂方法模式。

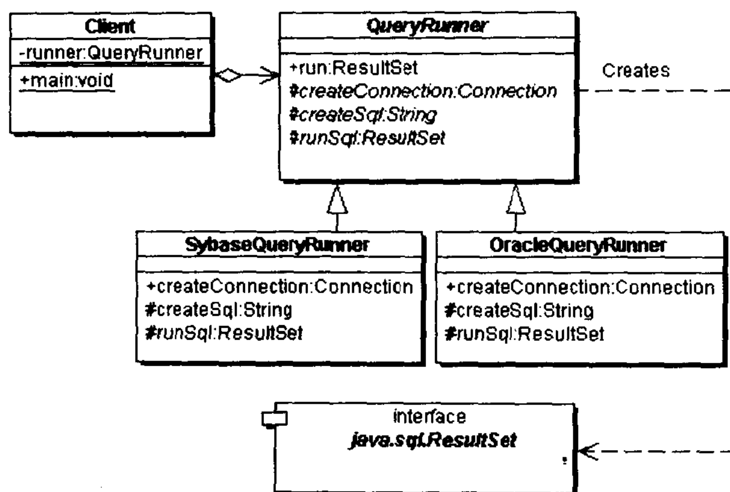
2. 这是不对的。一个工厂方法可以通过登记自己所创建的产品对象来实现循环提供相同的一些产品对象的功能。在享元模式中，就使用了这种工厂方法模式。

换言之，一个工厂方法所提供的产品对象不一定每次都是新的。

3. 不可以。工厂方法模式是创建模式，工厂方法的用意就是对对象创建过程的封装。虽然工厂方法不一定每一次都返还一个新的对象，但是工厂方法所返还的都应当是在工厂角色中被实例化的对象。

如果一个方法返还的对象是在另外一个对象中实例化的，那么这个方法不是工厂方法。

4. 下图所示就是这个查询运行器系统的设计图。



可以看出，这个系统是由一个客户端 Client（也就是产品的消费角色），一个抽象工厂角色 QueryRunner，两个具体工厂角色 SybaseQueryRunner 和 OracleQueryRunner，以及产品角色组成的。

对于客户端 Client 而言，系统的抽象产品角色是 ResultSet 接口，而具体产品角色就是 java.sql.Connection 所返还的具体 ResultSet 对象。但是如果仔细查看各个工厂角色就可以发现，createSql()方法和 createConnection()方法实际上也是工厂方法，它们的产品是 SQL 语句和 Connection 对象。

下面就是抽象工厂角色 QueryRunner 的源代码，如代码清单 20 所示。

代码清单 20：Java 抽象类 QueryRunner 的源代码

```
package com.javapatterns.factorymethod.query;
import java.sql.Connection;
import java.sql.ResultSet;
abstract public class QueryRunner
{
    public ResultSet run() throws Exception
    {
        Connection conn = createConnection();
        String sql = createSql();
        return runSql(conn, sql);
    }
    protected abstract Connection createConnection();
    protected abstract String createSql();
    protected abstract ResultSet runSql(Connection conn, String sql)
        throws Exception;
}
```

下面是具体工厂角色 SybaseQueryRunner 的源代码，如代码清单 21 所示。这个具体类实现了 Java 抽象类 QueryRunner 所声明的抽象方法。

代码清单 21：SybaseQueryRunner 的源代码

```
package com.javapatterns.factorymethod.query;
```

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
public class SybaseQueryRunner extends QueryRunner
{
    public Connection createConnection()
    {
        //示意性代码
        return null;
    }
    protected String createSql()
    {
        return "SELECT * FROM customers";
    }
    protected ResultSet runSql(Connection conn, String sql)
        throws Exception
    {
        Statement stmt = conn.createStatement();
        return stmt.executeQuery(sql);
    }
}
```

同样，OracleQueryRunner 也是一个具体工厂角色，它也是抽象工厂的子类，实现了 QueryRunner 所声明的抽象方法，如代码清单 22 所示。

代码清单 22：OracleQueryRunner 的源代码

```
package com.javapatterns.factorymethod.query;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
public class OracleQueryRunner extends QueryRunner
{
    public Connection createConnection()
    {
        //这下面仅仅是一个示意性的实现
        return null;
    }
    protected String createSql()
    {
        //示意性的 SQL 语句
        return "SELECT * FROM customers";
    }
    protected ResultSet runSql(Connection conn, String sql)
        throws Exception
    {
        Statement stmt = conn.createStatement();
        return stmt.executeQuery(sql);
```

```
    }
}
```

由于两个具体工厂子类 OracleQueryRunner 和 SybaseQueryRunner 分别处理与 Oracle 和 Sybase 数据库的连接和查询，因此，客户端就可以动态地决定何时采取哪一个具体工厂对象来创建 ResultSet 对象。

下面是一个示意性的客户端的源代码，如代码清单 23 所示。

代码清单 23：客户端 Client 类的示意性源代码

```
package com.javapatterns.factorymethod.query;
import java.sql.ResultSet;
public class Client
{
    private static QueryRunner runner;
    public static void main(String[] args)
        throws Exception
    {
        runner = new SybaseQueryRunner();
        ResultSet rs = runner.run();
    }
}
```

5．在上题给出的答案中，确实使用了模版方法模式。如果读者仔细查看一下系统设计图就会发现，在 QueryRunner 中，run()方法就是一个模版方法，这个方法代表一个顶级逻辑：返还查询的结果 ResultSet。

此逻辑在细节上分成几个步骤，分别由下面的基本方法完成：createSql()，createConnection()以及 runSql()，而这几个基本方法被当做剩余逻辑交给具体子类实现。

具体子类为两种数据库引擎提供了不同的实现，从而为一个顶级逻辑提供了不同的具体实现。

参考文献

[METSKER02] Steven J. Metsker. Design Patterns Java Workbook . published by Addison Wesley，2002

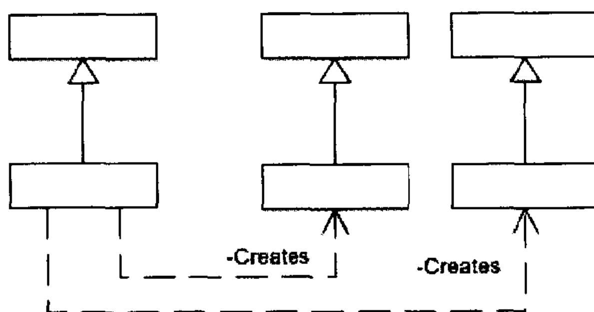
第 14 章　抽象工厂（Abstract Factory）模式

在阅读本章之前，请首先阅读本书的“简单工厂（Simple Factory）模式”以及“工厂方法（Factory Method）模式”两章。

14.1　引　　言

抽象工厂模式的用意

抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式的简略类图如下图所示。



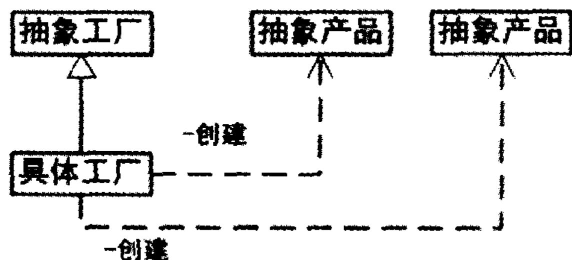
左边的等级结构代表工厂等级结构，右边的两个等级结构分别代表两个不同的产品的等级结构。

抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象。这就是抽象工厂模式的用意。

这是什么意思？相信很多读者会有这样的问题。为了说明抽象工厂模式的用意，不妨把它分成三段理解。

第一段

一个系统需要消费多个抽象产品角色，这些抽象产品角色可以用 Java 接口或者抽象 Java 类实现。读过本书的“工厂方法（Factory Method）模式”一章的读者可能会建议，既然客户端需要这些抽象产品角色的实例，为什么不使用一个工厂类负责创建这些角色的实例呢？工厂类负责创建抽象产品的实例描述如下图所示。

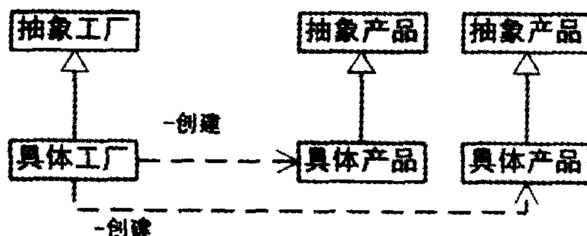


但是，正如上面所指出的，这些抽象产品角色是由 Java 接口或者抽象 Java 类实现的，而一个 Java 接口或者抽象 Java 类是不能实例化的。也就是说，上面的设计是不能成立的。

第二段

那么怎么满足系统的需求呢？

根据里氏代换原则，任何接收父类型的地方，都应当能够接收子类型。因此，实际上系统所需要的，仅仅是类型与这些抽象产品角色相同的一些实例，而不是这些抽象产品的实例。换言之，也就是这些抽象产品的具体子类的实例。工厂类负责创建抽象产品的具体子类的实例如下图所示。

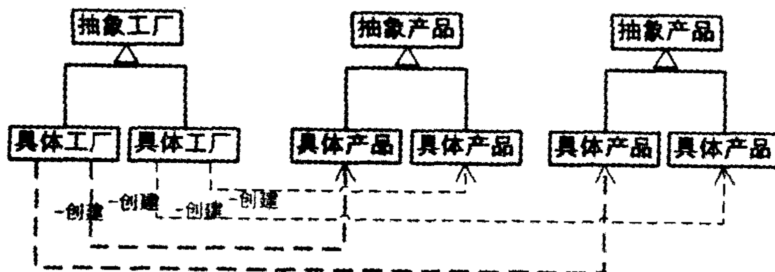


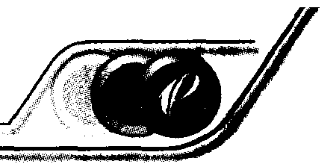
这就是抽象工厂模式用意的基本含义。

第三段

那么接下来的一个问题就是，如果每个抽象产品都有多于一个的具体子类的话，工厂角色怎么知道实例化哪一个子类呢？比如下面的类图中就给出了两个抽象产品，而每一个抽象产品都有两个具体产品。

抽象工厂模式提供两个具体工厂角色，分别对应于这两个具体产品角色。每一个具体工厂角色仅负责某一个具体产品角色的实例化。每一个具体工厂类负责创建抽象产品的某一个具体子类的实例，如下图所示。





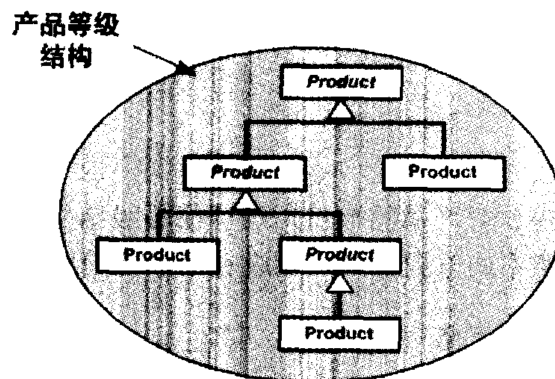
理解了这三个步骤，就不难理解“抽象工厂”这个名字的来源了。“抽象”来自“抽象产品角色”，而“抽象工厂”就是抽象产品角色的工厂。

14.2　问　　题

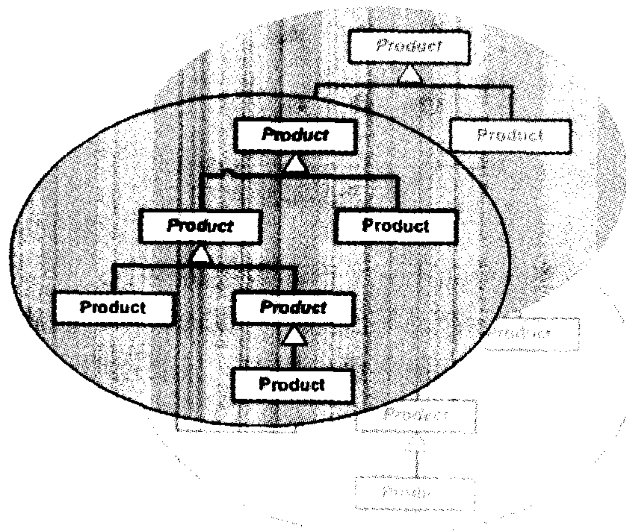
每一个模式都是针对一定问题的解决方案。正如前面所提到的，抽象工厂模式面对的问题是多个产品等级结构的系统设计。下面就从所面对的问题开始，将抽象工厂模式引进到系统设计中。

多个产品等级结构

抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。下图所示给出了一个产品等级结构。

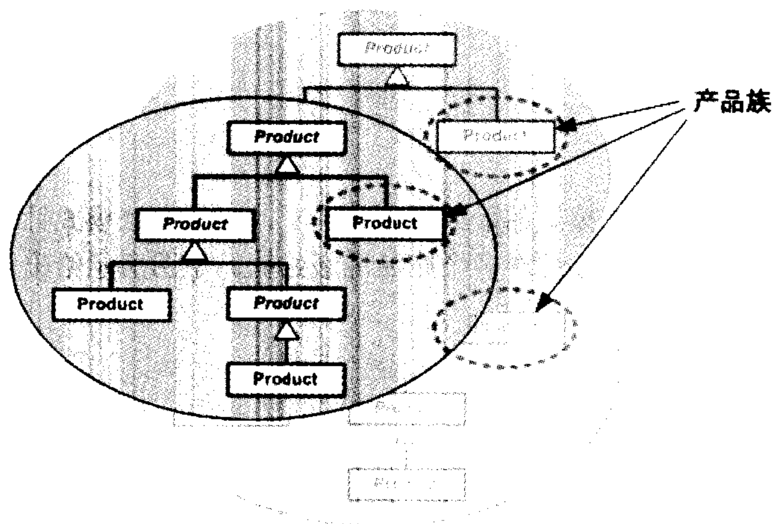


下图所示则给出了多个相平行的产品等级结构的例子。

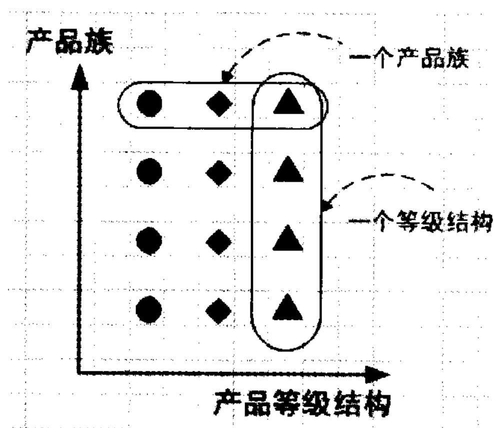


产品族

为了方便引进抽象工厂模式，特地引进一个新的概念：产品族（Product Family）。所谓产品族，是指位于不同产品等级结构中，功能相关联的产品组成的家族。如下图所示，箭头所指就是三个功能相关联的产品，它们位于三个不同的等级结构中的相同位置上，组成一个产品族。



显然，每一个产品族中含有产品的数目，与产品等级结构的数目是相等的。产品的等级结构和产品族将产品按照不同方向划分，形成一个二维的坐标系，如下图所示。



在坐标图中有四个产品族，分布于三个产品等级结构中。

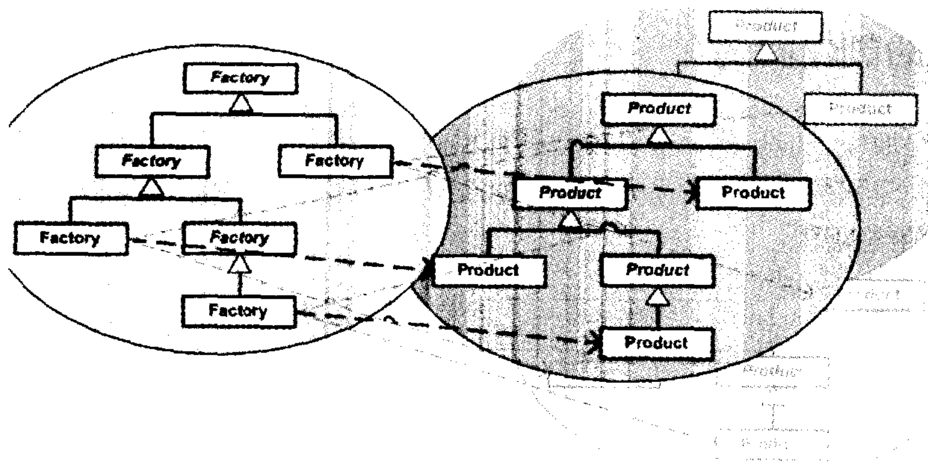
在上面的坐标图中，横轴表示产品等级结构，纵轴表示产品族。可以看出，图中一共有四个产品族，分布于三个不同的产品等级结构中。只要指明一个产品所处的产品族以及它所属的等级结构，就可以惟一地确定这个产品。

这样的坐标图，叫做相图。在一个相图中，坐标轴代表抽象的自由度，相图中两个坐标点之间的绝对距离并没有意义，有意义的是点与点的相对位置。

引进抽象工厂模式

上面所给出的三个不同的等级结构具有平行的结构。因此，如果采用工厂方法模式，就势必要使用三个独立的工厂等级结构来对付这三个产品等级结构。由于这三个产品等级结构的相似性，会导致三个平行的工厂等级结构。随着产品等级结构的数目的增加，工厂方法模式所给出的工厂等级结构的数目也会随之增加。

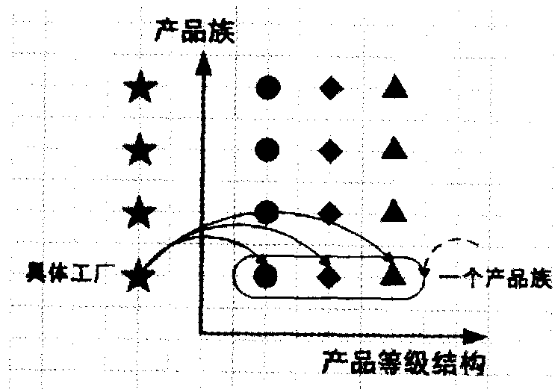
那么，是否可以使用同一个工厂等级结构来对付这些相同或者极为相似的产品等级结构呢？当然是可以的，而且这就是抽象工厂模式的好处。同一个工厂等级结构负责三个不同产品等级结构中的产品对象的创建，如下图所示，图中的虚线代表创建关系。



可以看出，一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象。显然，这时候抽象工厂模式比工厂方法模式更有效率。

应当指出的是，虽然大多数的文献都以一个含有两个层次（抽象和具体层次）的产品族作为讲解的例子，但在真实的系统中，产品族往往具有复杂的等级结构，就如同上面的图中所描述的一样，可以具有多于一个的抽象产品和很多的具体产品。

如果使用相图来描述的话，如下图所示。



在上面的相图中加入了具体工厂角色。可以看出，对应于每一个产品族都有一个具体工厂。而每一个具体工厂负责创建属于同一个产品族，但是分属于不同等级结构的产品。

14.3 抽象工厂模式的结构

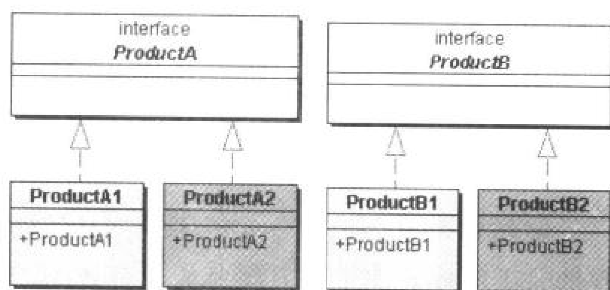
抽象工厂模式[GOF95]是对象的创建模式，它是工厂方法模式的进一步推广。

假设一个子系统需要一些产品对象，而这些产品又属于一个以上的产品等级结构。那么为了将消费这些产品对象的责任和创建这些产品对象的责任分割开来，可以引进抽象工厂模式。这样的话，消费产品的一方不需要直接参与产品的创建工作，而只需要向一个公用的工厂接口请求所需要的产品。

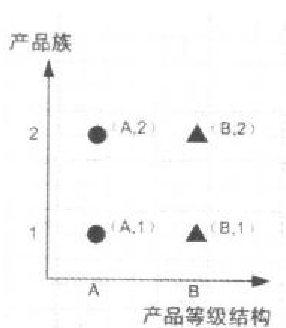
下面就以一个示意性的系统为例，说明这个模式的结构。

产品对象的创建问题

通过使用抽象工厂模式，可以处理具有相同（或者相似）等级结构的多个产品族中的产品对象的创建问题。如下图所示，图中就是两个具有相同等级结构的产品族 A 和产品等级结构 B 的结构图。



如果使用相图描述的话，会看到在相图上出现两个等级结构 A 和 B，以及两个产品族 1 和 2，如下图所示。

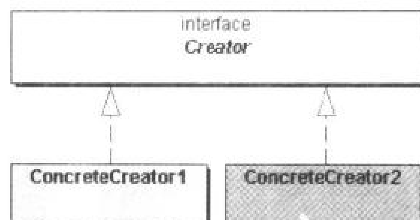


在上面的相图中，每一个坐标点都代表一个具体产品角色。可以看出，坐标点(A,1)，(A,2)，(B,1)和(B,2)分别对应于具体产品角色 ProductA1，ProductA2，ProductB1，ProductB2 等。

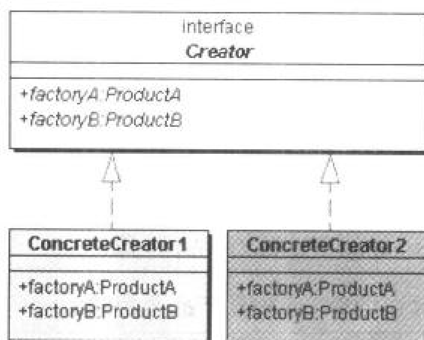
就像本章前面所谈到的一样，如果使用工厂方法模式处理的话，就必须要有两个独立的工厂族。由于这两个产品族的等级结构相同，因此，使用同一个工厂族也可以处理这两



个产品族的创建问题。后者就是抽象工厂模式，这样根据产品角色的结构图，就不难给出工厂角色的结构设计图，如下图所示。



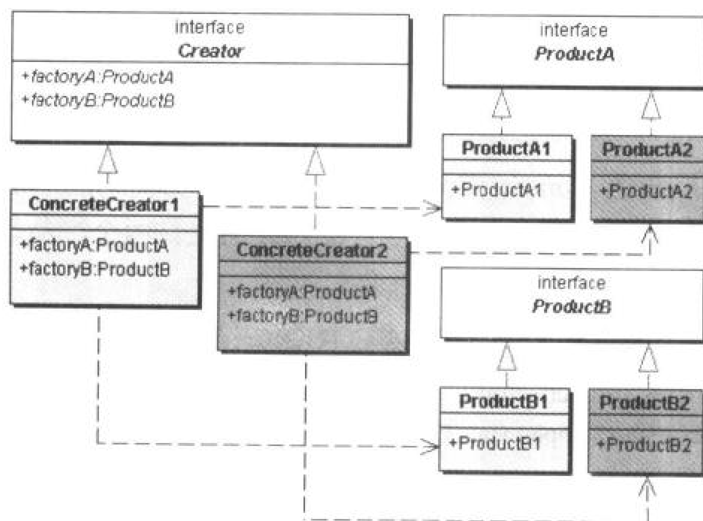
由于每个具体工厂角色都需要负责两个不同等级结构的产品对象的创建，因此每个工厂角色都需要提供两个工厂方法，分别用于创建两个等级结构的产品。既然每个具体工厂角色都需要实现这两个工厂方法，所以这种情况就具有一般性，不妨抽象出来，移动到抽象工厂角色 Creator 中加以声明。产品等级结构 A 和产品等级结构 B 的结构图如下图所示。



可以看出，每一个工厂角色都有两个工厂方法，分别负责创建分属不同产品等级结构的产品对象。

系统的设计

采用抽象工厂模式设计出的系统类图如下图所示。



从上图可以看出，抽象工厂模式涉及到以下的角色。

- 抽象工厂（AbstractFactory）角色：担任这个角色的是工厂方法模式的核心，它是与应用系统的商业逻辑无关的。通常使用 Java 接口或者抽象 Java 类实现，而所有的具体工厂类必须实现这个 Java 接口或继承这个抽象 Java 类。
- 具体工厂类（Concrete Factory）角色：这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。通常使用具体 Java 类实现这个角色。
- 抽象产品（Abstract Product）角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或它们共同拥有的接口。通常使用 Java 接口或者抽象 Java 类实现这一角色。
- 具体产品（Concrete Product）角色：抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。这是客户端最终需要的东西，其内部一定充满了应用系统的商业逻辑。通常使用具体 Java 类实现这个角色。

源代码

下面给出这个系统所有的源代码。

首先给出工厂角色的源代码，如代码清单 1 所示。可以看出，抽象工厂角色规定出两个工厂方法，分别提供两个不同等级结构的产品对象。

代码清单 1：抽象产品角色的源代码

```
package com.javapatterns.abstractfactory;
public interface Creator
{
   /**
    *  产品等级结构 A 的工厂方法
    */
    public ProductA factoryA();
   /**
    *  产品等级结构 B 的工厂方法
    */
    public ProductB factoryB();
}
```

下面给出具体工厂角色 ConcreteCreator1 的源代码，如代码清单 2 所示。这个具体工厂类实现了抽象工厂角色所要求的两个工厂方法，分别提供两个产品等级结构中的某一个产品对象。

代码清单 2：具体工厂类 ConcreteCreator1 的源代码

```
package com.javapatterns.abstractfactory;
public class ConcreteCreator1 implements Creator
{
   /**
```

```
     *  产品等级结构 A 的工厂方法
     */
     public ProductA factoryA()
     {
          return new ProductA1();
     }
    /**
     *  产品等级结构 B 的工厂方法
     */
     public ProductB factoryB()
     {
          return new ProductB1();
     }
}
```

一般而言，有多少个产品等级结构，就会在工厂角色中发现多少个工厂方法。每一个产品等级结构中有多少具体产品，就有多少个产品族，也就会在工厂等级结构中发现多少个具体工厂。

下面给出具体工厂角色 ConcreteCreator2 的源代码，如代码清单 3 所示。这个具体工厂类实现了抽象工厂角色所要求的两个工厂方法，分别提供两个产品等级结构中的另一个产品对象。

代码清单 3：具体工厂类 ConcreteCreator2 的源代码

```
package com.javapatterns.abstractfactory;
public class ConcreteCreator2 implements Creator
{
    /**
     *  产品等级结构 A 的工厂方法
     */
     public ProductA factoryA()
     {
          return new ProductA1();
     }
    /**
     *  产品等级结构 B 的工厂方法
     */
     public ProductB factoryB()
     {
          return new ProductB1();
     }
}
```

客户端需要的是产品，而不是工厂。在真实的系统中，产品类应当与应用系统的商业逻辑有密切关系。下面是产品等级结构 A 的抽象产品角色，在这个示意性的系统中，这个抽象产品角色是由一个 Java 接口实现的，如代码清单 4 所示。

代码清单 4：具体产品类 ProductA 的源代码

```
package com.javapatterns.abstractfactory;
public interface ProductA
{
}
```

下面是属于产品等级结构 A 的具体产品类 ProductA1 的源代码，如代码清单 5 所示。这个具体产品实现了产品等级结构 A 的抽象产品接口。

代码清单 5：具体产品类 ProductA1 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductA1 implements ProductA
{
    public ProductA1()
    {
    }
}
```

下面是同样属于产品等级结构 A 的具体产品类 ProductA2 的源代码，如代码清单 6 所示。这个具体产品也实现了产品等级结构 A 的抽象产品接口。

代码清单 6：具体产品类 ProductA2 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductA2 implements ProductA
{
    public ProductA2()
    {
    }
}
```

下面是产品等级结构 B 的抽象产品角色，这个抽象产品角色也是由一个 Java 接口实现的，如代码清单 7 所示。

代码清单 7：抽象产品角色 ProductB 的源代码

```
package com.javapatterns.abstractfactory;
public interface ProductB
{
}
```

可以看出这是一个标识接口（也就是没有声明任何方法的空接口，请参见本书的“专题：Java 接口”一章）。

下面是属于产品等级结构 B 的具体产品类 ProductB1 的源代码，如代码清单 8 所示。这个具体产品实现了产品等级结构 B 的抽象产品接口。

代码清单 8：具体产品类 ProductB1 的源代码

```
package com.javapatterns.abstractfactory;
```



```
public class ProductB1 implements ProductB
{
    /**
     *  构造子
     */
    public ProductB1()
    {
    }
}
```

下面是属于产品等级结构 B 的具体产品类 ProductB2 的源代码，如代码清单 9 所示。这个具体产品实现了产品等级结构 B 的抽象产品接口。

代码清单 9：具体产品类 ProductB2 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductB2 implements ProductB
{
    /**
     *  构造子
     */
    public ProductB2()
    {
    }
}
```

在本例中有两个产品等级结构，而每个产品等级结构中又恰好有两个产品，也就是有两个产品族。因此，工厂等级结构中就会出现两个具体工厂（对应于两个产品族）；而每个工厂类中又有两个工厂方法（对应于两个产品等级结构）。

在真实的系统中，产品等级结构的数目与每个产品等级结构中产品的数目（也就是产品族的数目）一般是不相等的。

14.4　在什么情形下应当使用抽象工厂模式

文献[GOF95]指出，在以下情况下应当考虑使用抽象工厂模式：

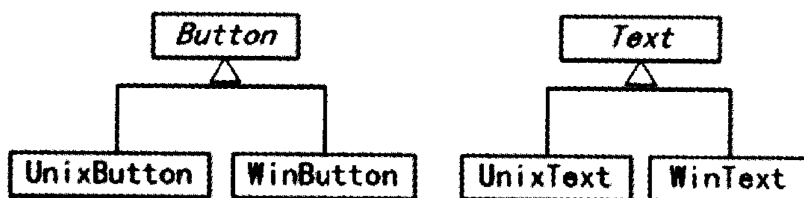
- 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。
- 这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品；（上面这一条叫做抽象工厂模式的原始用意。）
- 同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。
- 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

仔细思考一下，很多人都会问这样一个问题：为什么在第二条中说“系统只消费其中某一族的产品”呢？这实际上与抽象工厂模式的起源有关。

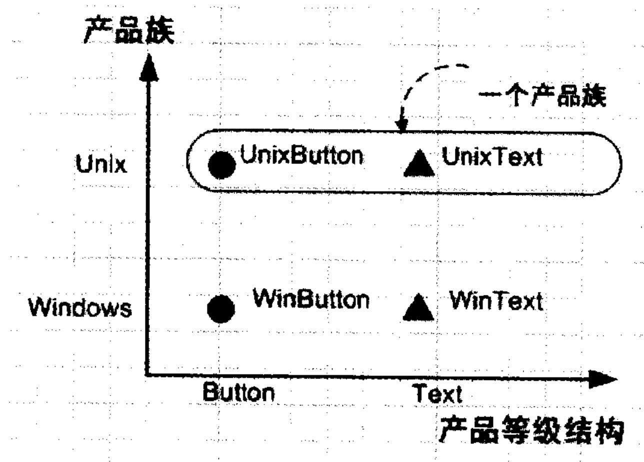
抽象工厂模式的起源

抽象工厂模式的起源或者说最早的应用，是用于创建分属于不同操作系统的视窗构件。比如，命令按键（Button）与文字框（Text）都是视窗构件，在 UNIX 操作系统的视窗环境和 Windows 操作系统的视窗环境中，这两个构件有不同的本地实现，它们的细节也有所不同。

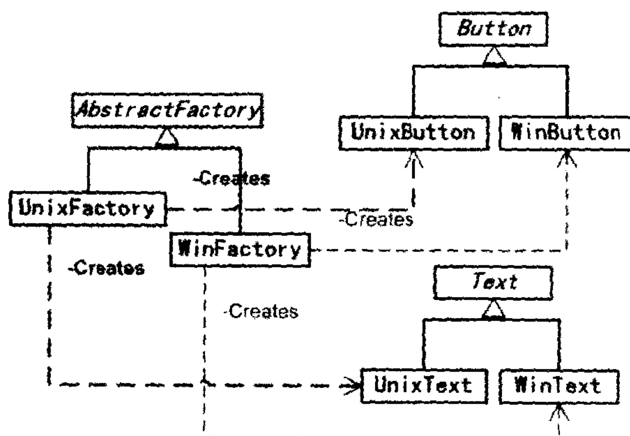
在每一个操作系统中，都有一个视窗构件组成的构件家族。在这里就是 Button 和 Text 组成的产品族。而每一个视窗构件都构成自己的等级结构，由一个抽象角色给出抽象的功能描述，而由具体子类给出不同操作系统下的具体实现，如下图所示。



可以发现在上面的产品类图中，有两个产品的等级结构，分别是 Button 等级结构和 Text 等级结构。同时有两个产品族，也就是 UNIX 产品族和 Windows 产品族。UNIX 产品族由 UnixButton 和 UnixText 产品构成；而 Windows 产品族由 WinButton 和 WinText 产品构成。其相图描述如下图所示。



系统对产品对象的创建需求由一个工厂的等级结构满足，其中有两个具体工厂角色，即 UnixFactory 和 WinFactory。UnixFactory 对象负责创建 Unix 产品族中的产品，而 WinFactory 对象负责创建 Windows 产品族中的产品。这就是抽象工厂模式的应用，抽象工厂模式的解决方案如下图所示。



显然，一个系统只能够在某一个操作系统的视窗环境下运行，而不能够同时在不同的操作系统上运行。所以，系统实际上只能消费属于同一个产品族的产品。

在现代的应用中，抽象工厂模式的使用范围已经大大扩大了，不再要求系统只能消费某一个产品族了。因此，读者可以不理会前面所提到的原始用意。

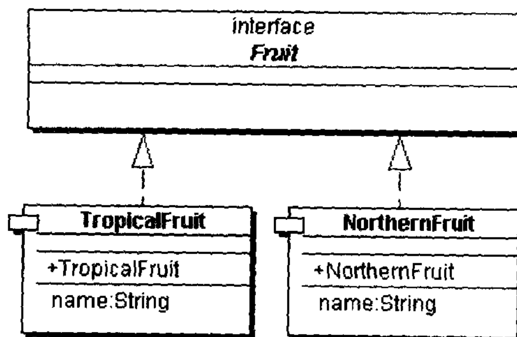
14.5 抽象工厂模式在农场系统中的实现

本节就考察一下如何扩展本书在“简单工厂（Simple Factory）模式”以及“工厂方法（Factory Method）模式”两章中所讨论过的农场系统。

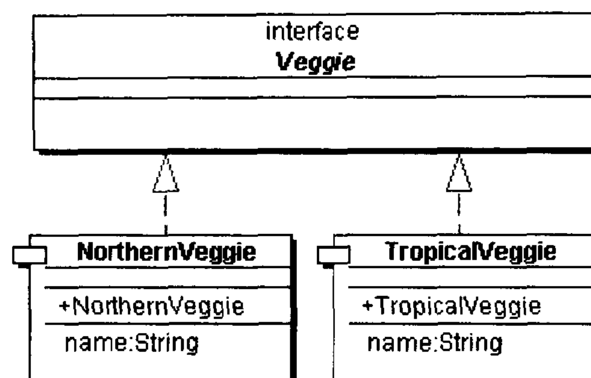
问题

本书在“简单工厂（Simple Factory）模式”与“工厂方法（Factory Method）模式”两章中曾经仔细讨论过一个农场公司从小到大的发展过程。而如今，农场公司再次面临新的大发展，一项重要的工作就是引进塑料大棚技术，在大棚里种植热带（Tropical）和亚热带的水果和蔬菜。

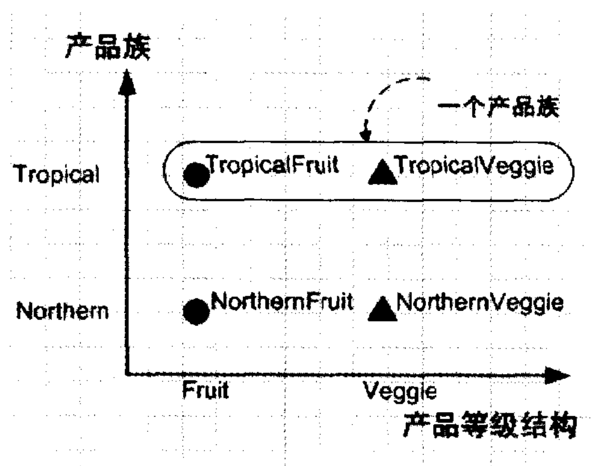
因此，在这个系统里面，产品分成两个等级结构：水果（Fruit）和蔬菜（Veggie）。下图所示就是水果（Fruit）的类图。



下图所示则是蔬菜（Veggie）的类图。

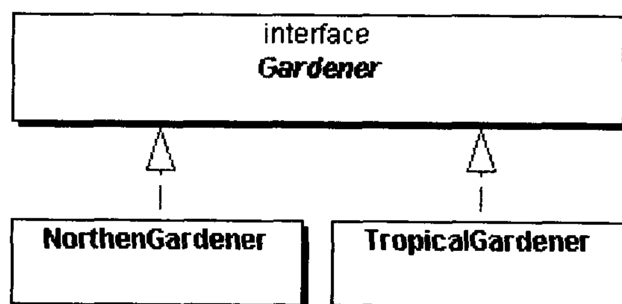


下图所示是描述这个系统的产品角色的相图。



可以看出，这个系统的产品可以分成两个等级结构：Fruit 和 Veggie，以及两个产品族：Tropical 和 Northern。坐标图上出现了四个坐标点，分别代表 TropicalFruit（热带水果）、TropicalVeggie（热带蔬菜）、NorthernFruit（北方水果）以及 NorthernVeggie（北方蔬菜）等四个产品。

显然可以使用一个工厂族来封装它们的创建过程。这个工厂族的等级结构应当与产品族的等级结构完全平行，园丁等级结构的类图如下图所示。

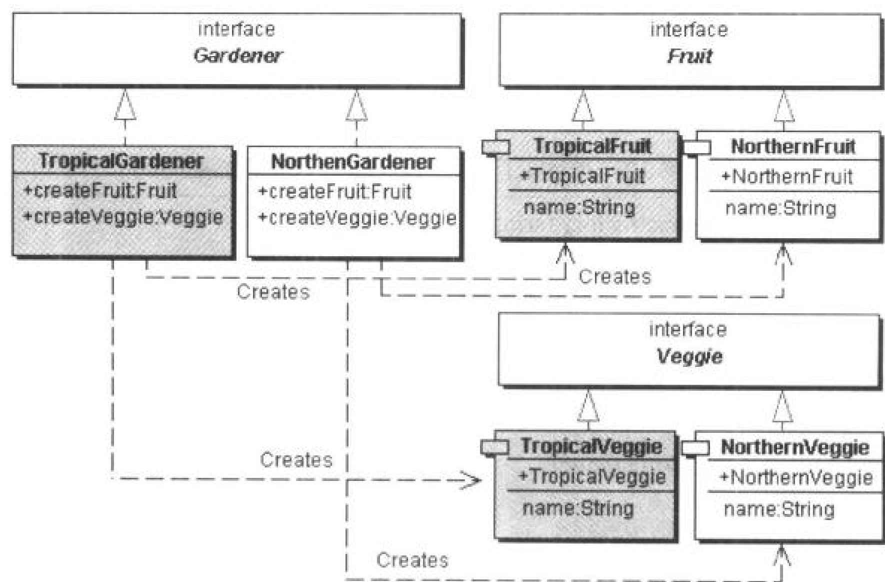


系统所需要的是产品的实例，而工厂则是对产品创建过程的封装。



系统设计

与抽象工厂模式的各个角色相对照，不难发现，所谓各个园丁其实就是各个工厂角色，而蔬菜和水果角色则是产品角色。将抽象工厂模式应用于农场系统中，系统的设计图如下图所示。



种在田间的北方作物与种在大棚的热带作物都是系统的产品，它们分属于两个产品族。显然，北方作物是要种植在一起的，而大棚作物是要另外种植在一起的。这些分别体现在系统的设计上，就正好满足了使用抽象工厂模式的第三个条件。

首先，Gardener 接口是一个没有任何方法的 Java 接口。读过本书的“专题：Java 接口”一章的读者可以看出，这是一个标识接口，如代码清单 10 所示。

代码清单 10：接口 Gardener

```
public interface Gardener { }
```

NorthernGardener 和 TropicalGardener 均是抽象工厂类 Gardener 的具体子类，也就是说它们全都是具体工厂类，如代码清单 11 所示。

代码清单 11：具体工厂类 NorthernGardener 的源代码

```
package com.javapatterns.abstractfactory.farm;
public class NorthernGardener implements Gardener
{
    /**
     * 水果的工厂方法
     */
    public Fruit createFruit(String name)
    {
        return new NorthernFruit(name);
```

```
    }
    /**
     *  蔬菜的工厂方法
     */
    public Veggie createVeggie(String name)
    {
        return new NorthernVeggie(name);
    }
}
```

另一个具体工厂类 TropicalGardener 的源代码如代码清单 12 所示。

代码清单 12：具体工厂类 TropicalGardener 的源代码

```
package com.javapatterns.abstractfactory.farm;
public class TropicalGardener implements Gardener
{
    /**
     *  水果的工厂方法
     */
    public Fruit createFruit(String name)
    {
        return new TropicalFruit(name);
    }
    /**
     *  蔬菜的工厂方法
     */
    public Veggie createVeggie(String name)
    {
        return new TropicalVeggie(name);
    }
}
```

显然 Veggie 是一个标识接口，如代码清单 13 所示。

代码清单 13：接口 Veggie

```
public interface Veggie {}
```

北方的蔬菜 NorthernVeggie 应当实现 Veggie 接口，其源代码清单 14 所示。

代码清单 14：具体产品类 NorthernVeggie 的源代码

```
package com.javapatterns.abstractfactory.farm;
public class NorthernVeggie implements Veggie
{
    private String name;
    public NorthernVeggie(String name)
    {
    }
    public String getName()
```



```
        {
            return name;
        }
        public void setName(String name)
        {
            this.name = name;
        }
    }
```

热带蔬菜 TropicalVeggie 也应当实现 Veggie 接口，其源代码如代码清单 15 所示。

代码清单 15：具体产品类 TropicalVeggie 的源代码

```
public class TropicalVeggie implements Veggie
{
    private String name;

   /**
    *  构造子
    */
    public TropicalVeggie(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
}
```

抽象产品角色 Fruit 有一个 Java 接口实现，它的源代码如代码清单 16 所示。

代码清单 16：抽象产品角色 Fruit 的源代码

```
package com.javapatterns.abstractfactory.farm;
public interface Fruit
{
}
```

而北方水果 NorthernFruit 类则实现了抽象水果接口 Fruit，其源代码如代码清单 17 所示。

代码清单 17：抽象产品角色 NorthernFruit 的源代码

```
package com.javapatterns.abstractfactory.farm;
public class NorthernFruit implements Fruit
{
```

```
        private String name;
        public NorthernFruit(String name)
        {
        }
        public String getName()
        {
            return name;
        }
        public void setName(String name)
        {
            this.name = name;
        }
    }
```

同样，热带水果 TropicalFruit 类也实现了抽象水果接口 Fruit，其源代码如代码清单 18 所示。

代码清单 18：抽象产品角色 TropicalFruit 的源代码

```
package com.javapatterns.abstractfactory.farm;
public class TropicalFruit implements Fruit
{
    private String name;
    public TropicalFruit(String name)
    {
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
}
```

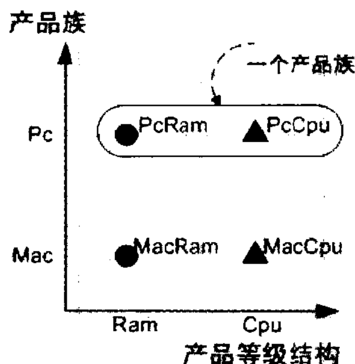
在使用时，客户端只需要创建具体工厂的实例，然后调用工厂对象的工厂方法，就可以得到所需要的产品对象。

14.6 抽象工厂模式的另一个例子

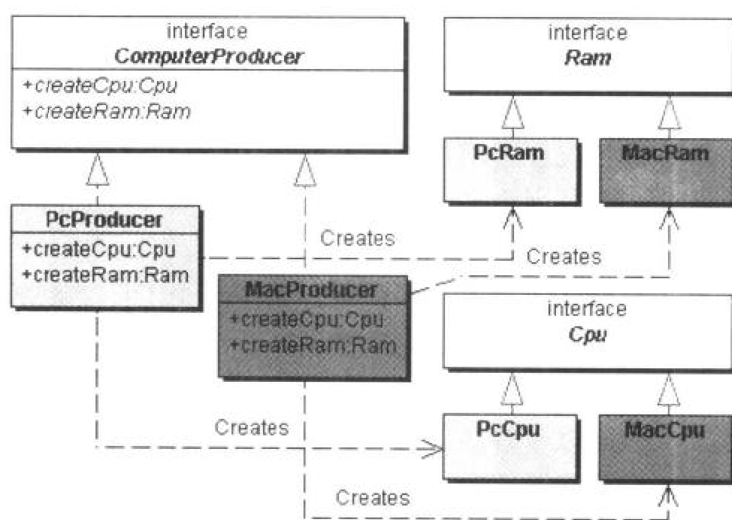
这个例子描述微型计算机配件的生产。这个系统所需要的产品族有两个，一个系列是 PC，或称 IBM 及 IBM 克隆机系列；另一个系列是 MAC，或称 MacIntosh 系列。产品等级结构也有两个，一个是 RAM，另一个是 CPU。

如果使用相图描述的话，应当是如下图所示的样子。





显然，这个系统应该使用抽象工厂模式，而不是工厂方法模式，因为后者适合于处理只有一个产品等级结构的情形。抽象工厂模式应用于微型计算机配件的生产系统中，如下图所示。



两种不同的背景颜色可以区分两个不同的产品族，及其每一个产品族所对应的具体工厂类。

从图中可以看出，每一个工厂角色都提供两个工厂方法，分别对应于两个不同的抽象产品。有多少抽象产品，就会有多少的工厂方法。限于篇幅，不再提供这个系统的详细讨论。但是，读者可以在本章后面的问答题中找到更多的关于这个系统的内容。

14.7　“开–闭”原则

“开–闭”原则要求一个软件系统可以在不修改原有代码的情况下，通过扩展达到增强其功能的目的。对于一个涉及到多个产品等级结构和多个产品族的系统，其功能的增强不外乎两个方面：

- 增加新的产品族。

● 增加新的产品等级结构。

那么抽象工厂模式是怎样支持这两方面功能的增强的呢？

增加新的产品族

在产品等级结构的数目不变的情况下，增加新的产品族，就意味着在每一个产品等级结构中增加一个（或者多个）新的具体（或者抽象和具体）产品角色。

由于工厂等级结构是与产品等级结构平行的登记机构，因此，当产品等级结构有所调整时，需要将工厂等级结构做相应的调整。现在产品等级结构中出现了新的元素，因此，需要向工厂等级结构中加入相应的新元素就可以了。

换言之，设计师只需要向系统中加入新的具体工厂类就可以了，没有必要修改已有的工厂角色或者产品角色。因此，在系统中的产品族增加时，抽象工厂模式是支持“开-闭”原则的。

增加新的产品等级结构

在产品族的数目不变的情况下，增加新的产品等级结构。换言之，所有的产品等级结构中的产品数目不会改变，但是现在多出一个与现有的产品等级结构平行的新的产品等级结构。

要做到这一点，就需要修改所有的工厂角色，给每一个工厂类都增加一个新的工厂方法，而这显然是违背“开-闭”原则的。换言之，对于产品等级结构的增加，抽象工厂模式是不支持“开-闭”原则的。

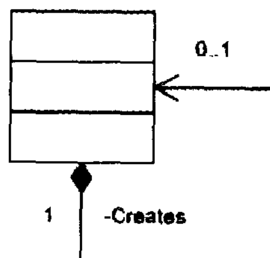
综合起来，抽象工厂模式以一种倾斜的方式支持增加新的产品，它为新产品族的增加提供方便，而不能为新的产品等级结构的增加提供这样的方便。

14.8 相关的模式与模式的实现

抽象工厂模式与一些其他的设计模式有密切的关系，因而在 Java 语言中实现抽象工厂模式时，有下面一些值得注意的地方。

具体工厂类与单例模式

具体工厂类可以设计成单例类。单例模式的简略类图如右图所示。



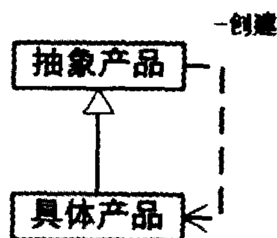
一个单例类只有一个实例，它自己向外界提供自己惟一的实例。关于单例类的知识，请见本书的“单例（Singleton）模式”一章。

很显然，在农场系统中，只需要 NorthernGardener 和

TropicalGardener 的一个实例就可以了。在计算机生产的例子中，PcProducer 和 MacProducer 也分别只需要一个实例。因此，这两个具体工厂角色都可以设计成单例类。

工厂的工厂（之一）

在本书的“简单工厂（Simple Factory）模式”一章中，曾经谈到在简单工厂模式中，工厂角色可以与抽象产品角色合并，并以 java.util.DateFormat 为例子讲解这一做法。如右图所示。

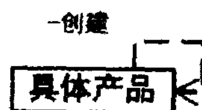


这种做法可以应用到工厂等级结构中去，只是注意在这里“产品”不是产品角色，而是具体工厂角色。

在抽象工厂模式中，抽象工厂类可以配备静态方法，以返还具体工厂的实例。具体地讲，抽象工厂角色可以配备一个静态方法，这个方法按照参量的值，返回所对应的具体工厂的实例。静态方法的返还类型是抽象工厂类型，这样可以在多态性的保证之下，允许静态工厂方法自行决定哪一个具体工厂符合要求。

工厂的工厂（之二）

在本书的“简单工厂（Simple Factory）模式”一章中，还曾经谈到在简单工厂模式中，工厂角色可以与具体工厂角色合并，如右图所示。



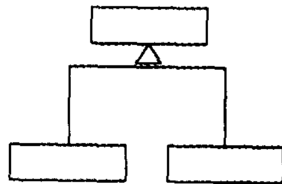
这种做法也可以应用到工厂等级结构中去，只是注意在这里“产品”不是产品角色，而是具体工厂角色。

这就意味着要为每一个具体工厂类配备一个静态方法，而其返还类型是该具体工厂类自己。

通过工厂方法模式或者原始模型模式实现

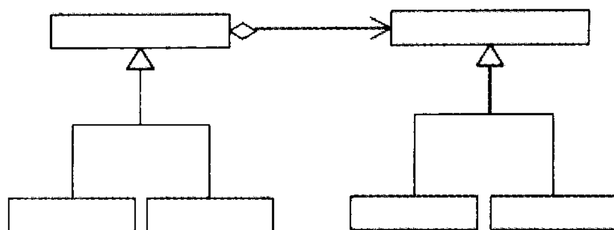
在本章的所有例子中，抽象工厂模式都是通过工厂方法模式实现的。换言之，每一个工厂角色都配有一个工厂方法，这个方法负责调用产品的构造子，将产品角色实例化。

抽象工厂模式完全可以使用原始模型模式而不是工厂方法模式实现。读者可以参阅本书的“原始模型（Prototype）模式”一章，这里不再赘述。原始模型模式的简略类图如右图所示。



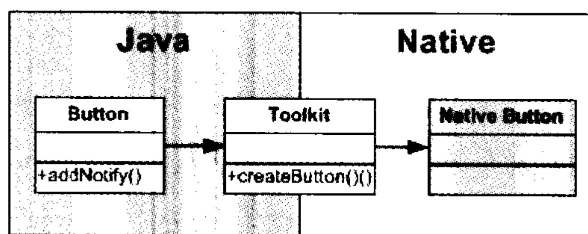
桥梁模式

抽象工厂模式可以为桥梁模式提供某一个等级结构的创建功能，抽象模式可以与桥梁模式一同使用。桥梁模式的简略类图如下图所示。



在 Java 的 AWT 库中，定义了两套平行的等级结构：一套是 Java 的构件，以 Component 为超类，另一套是所谓的 Peer 构件，以 ComponentPeer 为超类。Java 构件向 Java 程序提供一套与操作系统无关的、统一的构件接口；而 Peer 构件则处理底层的、与操作系统密切相关的功能。

在这两个等级结构之间的，是 java.awt.Toolkit 类；这个抽象类在不同的操作系统中有不同的具体子类，并为每一个 Peer 构件提供了相应的工厂方法，以创建并且返还一个 Peer 构件的实例。如下图所示，图中显示了 Java 构件 Button 通过调用 Toolkit 工厂对象创建一个 ButtonPeer 对象的情况。



上面是 Button 和 ButtonPeer 的通信图。Button 是一个 Java 构件，通过 Toolkit 对象与本地 Peer 构件通信。在这个结构中，Toolkit 就是抽象工厂角色，它的具体实现是具体工厂角色；而各个 Peer 对象就是具体产品角色。Peer 对象根据自己的分类形成继承的等级结构；而根据操作系统划分成不同的产品族。

这两个等级结构之间通过委派进行通信，形成桥梁模式的结构。详细的讨论请参见本章后面的附录“Java AWT 的 Peer 架构与抽象工厂模式”，以及本书的“桥梁（Bridge）模式”一章。

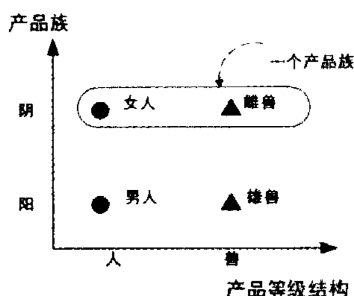
14.9　女娲造万物的故事

《说文解字》在解释“娲”时，云：“古之神圣女，化万物者”。换言之，女娲不仅仅造了人，而且造了世间万物，这也包括各种动物。因此，女娲一定是把举绳造人的方法推广应用到了创造各种动物身上。可以想到，女娲把绳子搅到泥水里，然后把沾满泥水的绳子凭空一甩，甩出的泥点，像人的变成了人，像各种其他动物的泥点则变成了其他的动物。女娲的阴绳造出的是女人和雌动物，阳绳造出的是男人和雄动物。

读者可以看出，女娲造物用的是抽象工厂模式。在这个故事里面，女娲的“产品”有两个划分方法：一是按照“产品”是人还是兽来划分，二是按照“产品”是男女、雌雄来



划分。女娲的绳子按照阴、阳划分，产品则按照人、兽划分。女娲造万物系统里阴、阳两个等级结构和人、兽两个产品族的类图如下图所示。



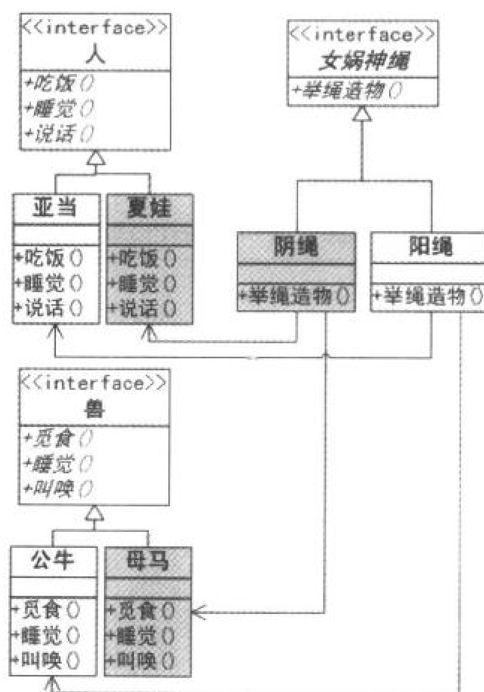
换言之，女娲的故事里面有如下的抽象角色：

- “神绳”接口作为抽象工厂角色。
- “人”接口作为人类抽象角色。在女娲造人之前，一定在头脑里有了一个对人的样子的想像，这个想像就是对人的抽象。
- “兽”接口作为兽类抽象角色。在女娲造各种野兽之前，一定在头脑里有了一个对野兽样子的想像，这个想像就是对野兽的抽象。

还有继承自抽象角色的具体角色：

- 阴绳、阳绳继承自“神绳”接口，是具体的绳子类。
- 亚当、夏娃继承自“人”接口，是具体的人类。本书选用亚当和夏娃为例，是为了彰显中西合璧的妙处。
- 公牛、母马继承自“兽”接口，是具体的野兽类，它必须符合女娲对野兽的设想。

将抽象工厂模式应用于女娲造万物的模拟系统设计中。如下图所示，两种颜色代表阴、阳两种系列。



显然，产品类有两个族，为阴和阳（男和女或公和母）；有两个产品等级结构，为人和兽。这种多于一个产品等级结构的情况是需要抽象工厂模式的关键原因。

问答题

1. 如上面的讨论，抽象工厂类可以配备一个静态方法，按照参量的值返回所对应的具体工厂。请把微型计算机生产系统的抽象工厂类按照这一方案改造，给出类图和源代码。

2. 如上面的讨论，具体工厂类可以设计成单例类。请在第 1 题的基础上把微型计算机生产系统的具体工厂类按照这一方案改造，给出 UML 类图和源代码。

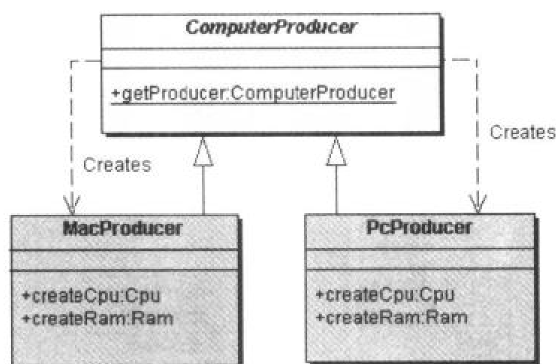
3. 请问相图与 UML 图是什么关系？

4. 请使用相图描述一下工厂方法模式。

5. 请使用相图描述一下简单工厂模式。

问答题答案

1. 微型计算机生产系统的抽象工厂原本是接口，现在需要改造成抽象类。如下图所示，其 ComputerProducer 的类名为斜体，表明该类是抽象的；而 getProducer()的下划线，表明该方法是静态的。



经过单例模式改造的 ComputerProducer 的源代码如代码清单 19 所示。

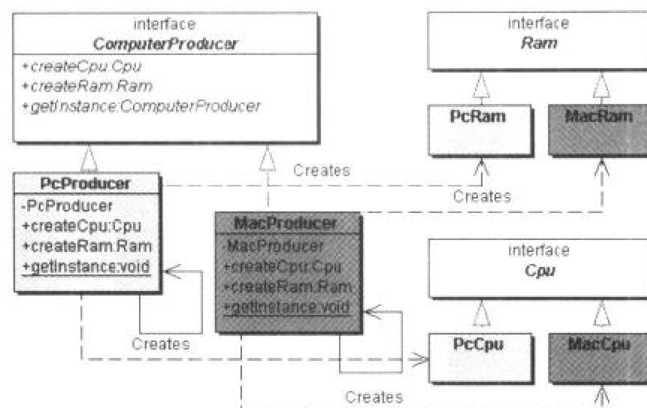
代码清单 19：抽象类 ComputerProducer 的源代码

```
public class ComputerProducer
{
    public static ComputerProducer getProducer(String which)
    {
        if (which.equalsIgnoreCase("Pc"))
        {
            return new PcProducer();
        }
        else if (which.equalsIgnoreCase("Mac"))
        {
            return new MacProducer();
        }
```



```
        else
        {
            return null;
        }
    }
}
```

2. 本题答案是建立在第 1 题基础之上的。如下图所示，三种不同的背景颜色可以区分抽象工厂类，两类产品族，及其对应的具体工厂类。



ComputerProducer 类为斜体，表明该类是抽象的，而 getInstance()的下划线表明该方法是静态的。MacProducer 和 PcProducer 的构造子是私有的，因此这两个类必须自己将自己实例化。

抽象工厂类 ComputerProducer 的源代码如代码清单 20 所示。

代码清单 20：抽象工厂类 ComputerProducer 的源代码

```
abstract public class ComputerProducer
{
    public static ComputerProducer getInstance(String which)
    {
        if (which.equalsIgnoreCase("Pc"))
        {
            return PcProducer.getInstance();
        }
        else if (which.equalsIgnoreCase("Mac"))
        {
            return MacProducer.getInstance();
        }
    }
}
```

具体工厂类 MacProducer 的源代码如代码清单 21 所示。

代码清单 21：具体工厂类 MacProducer 是单例类

```
public class MacProducer extends ComputerProducer
{
```

```
    private static MacProducer m_MacProducer =
        new MacProducer();
   /**
    *  私有的构造子，保证外界不能直接实例化
    */
    private MacProducer()
    {
    }
   /**
    *  工厂方法，返还产品实例
    */
    public Cpu createCpu()
    {
        return new MacCpu();
    }
   /**
    *  工厂方法，返还产品实例
    */
    public Ram createRam()
    {
        return new MacRam();
    }
   /**
    *  静态工厂方法，返还单例实例
    */
    public static MacProducer getInstance()
    {
        return m_MacProducer;
    }
}
```

读过本书“单例（Singleton）模式”一章的读者应当知道，这里使用的单例类实现方法是饿汉式方法。

具体工厂类 PcProducer 的源代码如代码清单 22 所示。

代码清单 22：具体工厂类 MacProducer 是单例类

```
public class PcProducer extends ComputerProducer
{
    private static PcProducer m_PcProducer =
        new PcProducer();
   /**
    *  私有的构造子，保证外界不能直接实例化
    */
    private PcProducer()
    {
    }
```

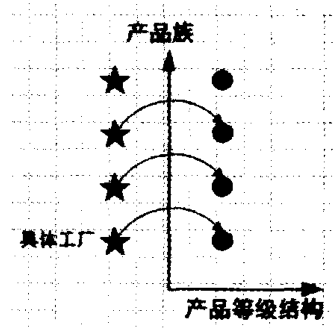
```
    /**
     *  工厂方法，返还产品实例
     */
     public Cpu createCpu()
     {
          return new PcCpu();
     }
    /**
     *  工厂方法，返还产品实例
     */
     public Ram createRam()
     {
          return new PcRam();
     }

    /**
     *  静态工厂方法，返还单例实例
     */
     public static PcProducer getInstance()
     {
          return m_PcProducer;
     }
}
```

使用的单例类实现方法是饿汉式方法。各产品类没有变化，因此不再重复。

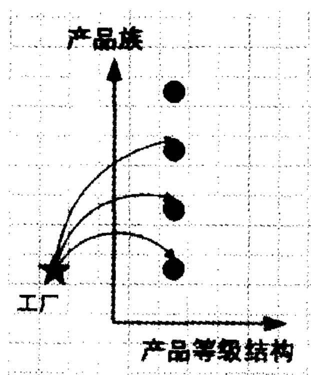
3. 相图与 UML 图没有关系。

4. 在工厂方法模式中，只有一个产品的等级结构，工厂的等级结构与产品的等级结构是平行的。因此，不难给出这个模式的相图，如右图所示。



图中的左侧有一列具体工厂，右侧有一列具体产品，从工厂到产品是创建关系。

5. 在简单工厂模式中，只有一个工厂角色和 ·个产品的等级结构。因此不难给出这个模式的相图，如下图所示。



14.10 附录：Java AWT 的 Peer 架构与抽象工厂模式

在“抽象工厂（Abstract Factory）模式”一章中的“抽象工厂模式的起源”一节中，本书使用示意性的结构讲解了怎样将抽象工厂模式应用到多个操作系统的视窗构件的创建中。实际上，在 Java 语言的 AWT 库里，确实使用了抽象工厂模式创建分属于不同操作系统的 Peer 构件。

本节就对 AWT 这个库，特别是库中的 Toolkit 类及其子类做一个考察，以说明抽象工厂模式是怎么应用到 AWT 库中的。

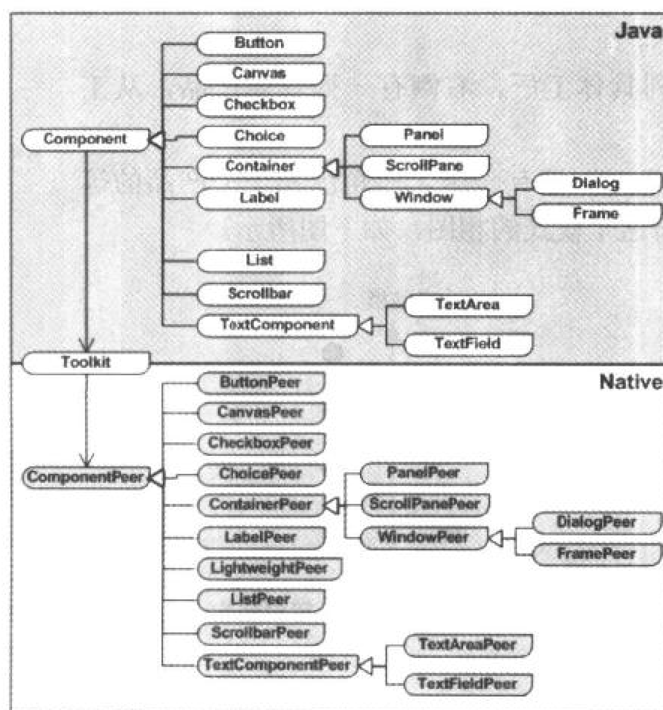
Peer 架构

就从 Java 用户界面的视感谈起吧。

一个有用户界面的 Java 程序总是显示其所在操作系统的视感（Look and Feel）；而 Java 程序内部却总是对一个统一的构件接口编程。这是由于在同一个构件接口后面，还存在着一层软件结构，叫做 Peer 接口。在这个接口和 Java 构件接口之间，存在着 Toolkit 接口，负责 Peer 对象的创建。这种三重接口的架构，叫做 Peer 架构。显然，Peer 架构是桥梁模式的应用。

每有一个 Java 视窗构件，就有一个对应的 Peer 接口，而这个接口在不同的操作系统中有不同的实现。在运行时，Peer 架构会自行产生一个对应于当前操作系统的 Toolkit 对象。在运行时，Peer 架构会自行调用这个 Toolkit 对象，创建出所需要的 Peer 对象。

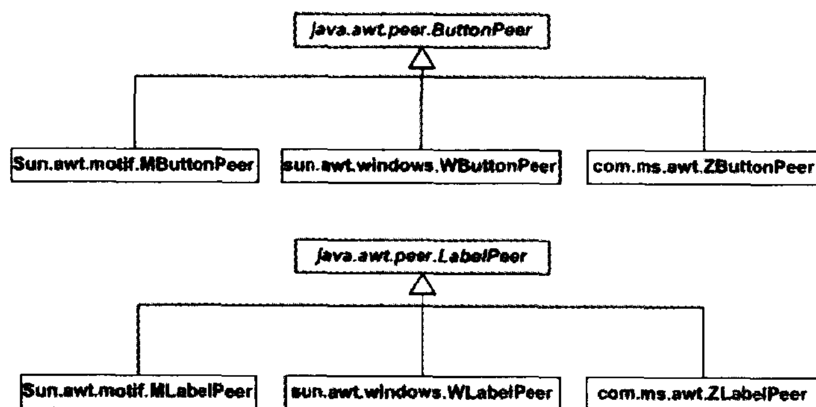
如下图所示，Component 类型和 ComponentPeer 类型通过 Toolkit 对象相互通信。



上面的结构图分成上下两个部分，下方的部分给出的就是 java.awt.peers 库中所有的 Peer 接口。关于 Peer 接口与 Peer 架构，读者可以参考阅读本书的“桥梁（Bridge）模式”一章。

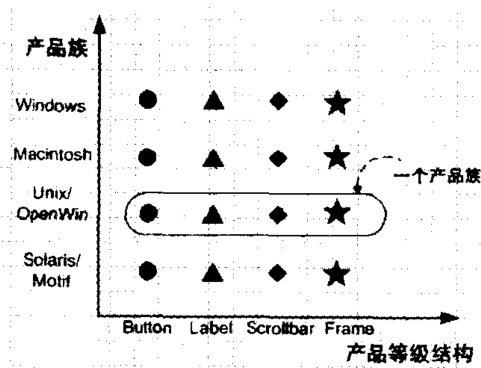
产品角色与 Peer 对象

如下图所示，图中显示了两个产品等级结构（即 ButtonPeer 和 LabelPeer 等级结构），以及三个产品族，即 Windows 产品族，Solaris/Motif 产品族和 Unix/OpenWin 产品族。



上面的图显示了 Peer 构件形成产品的等级结构。

如果读者认真看一看 JDK 所带的源代码库的话，可以看到更多的产品等级结构，包括 Window、Frame、Scrollbar 等；也会看到更多的产品族，比如 Macintosh 产品族等。换言之，每一个支持 Java 的操作系统都必须为所有的视窗构件的等级结构提供一个相应的实现，如下图所示。



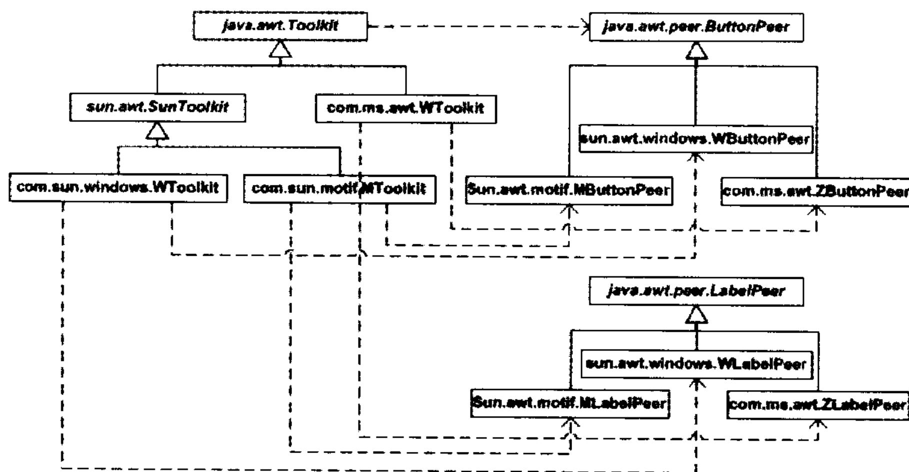
上面是 Peer 构件形成产品的相图。显然，由于一个程序不可能同时在两个操作系统中运行，因此在每一个时刻只可能有一个产品族的产品被创建和消费。这就是说，抽象工厂模式的原始用意在这里是被满足的。

实际上，要想收集到所有的产品族的产品（如 Zbutton、Wbutton 和 MButton 等，分别是为 Windows、Unix/OpenWin 和 Solaris/Motif 准备的 Peer 构件）并不容易。因为 JDK 是分别为不同的操作系统准备的，所以在每一个版本的 JDK 中，只能找到相关操作系统

的 Peer 构件。

工厂角色与 Toolkit

如下图所示，图中显示了一个工厂等级结构，这个等级结构的超类是 java.awt.Toolkit。每一个支持 Java 的操作系统都必须提供一个 Toolkit 的具体子类。读者可以看出，图中显示的三个 Toolkit 的具体子类分别对应于三个不同的操作环境。而如果添加上 Macintosh 的话，就应当再加上一个 Toolkit 的具体子类（应当是 sun.awt.macos.MToolkit）。



从上面的图可以看出各个 Toolkit 对象位于工厂等级结构中，负责创建 ButtonPeer 等级结构和 LabelPeer 等级结构中的对象。

虽然 Toolkit 是 Java API 中使用较为频繁的一个，但是在大多数情况下，它是在背后运行的，普通的 Java 引用程序很少需要直接创建一个 Toolkit 类的实例。

如果程序需要一个 Toolkit 对象，就应当调用它的静态工厂方法 getDefaultToolkit()，或者 Component.getToolkit()方法，以得到一个正确的 Toolkit 对象。

桥梁模式的应用

在 Peer 架构中还使用了桥梁模式。请参见本书的“桥梁（Bridge）模式”一章。

参考文献

[ZUKOWSKI97] John Zukowski. Java AWT Reference. published by O'Reilly, 1997

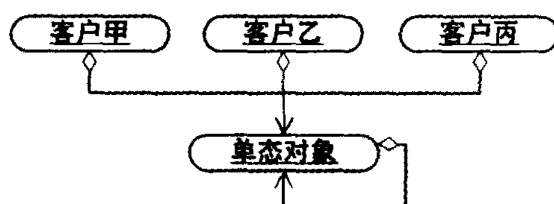
第 15 章　单例（Singleton）模式

作为对象的创建模式[GOF95]，单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

15.1　引　　言

单例模式的要点

显然单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个事例；三是它必须自行向整个系统提供这个实例。在下图所示的对象图中，有一个“单例对象”，而“客户甲”、“客户乙”和“客户丙”是单例对象的三个客户对象。可以看到，所有的客户对象共享一个单例对象，而且从单例对象到自身的连接线可以看出，单例对象持有对自己的引用。



资源管理

一些资源管理器常常设计成单例模式。

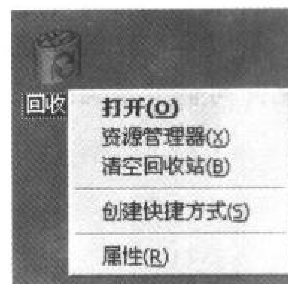
在计算机系统中，需要管理的资源包括软件外部资源，譬如每台计算机可以有若干个打印机，但只能有一个 Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干个传真卡，但是只应该有一个软件负责管理传真卡，以避免出现两份传真作业同时传到传真卡中的情况。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。

需要管理的资源包括软件内部资源，譬如，大多数的软件都有一个（甚至多个）属性（properties）文件存放系统配置。这样的系统应当由一个对象来管理一个属性文件。

需要管理的软件内部资源也包括譬如负责记录网站来访人数的部件，记录软件系统内部事件、出错信息的部件，或是对系统的表现进行检查的部件等。这些部件都必须集中管理，不可政出多头。

这些资源管理器构件必须只有一个实例，这是其一；它们必须自行初始化，这是其二；允许整个系统访问自己，这是其三。因此，它们都满足单例模式的条件，是单例模式的应用。

一个例子：Windows 回收站



Windows 9x 以后的视窗系统中都有一个回收站，如右图所示，就显示了 Windows 2000 的回收站。

在整个视窗系统中，回收站只能有一个实例，整个系统都使用这个惟一的实例，而且回收站自行提供自己的实例。因此，回收站是单例模式的应用。

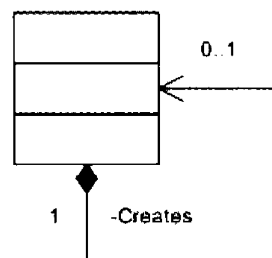
双重检查成例

在本章最后的附录里研究了双重检查成例。双重检查成例与单例模式并无直接的关系，但是由于很多 C 语言设计师在单例模式里面使用双重检查成例，所以这一做法也被很多 Java 设计师所模仿。因此，本书在附录里提醒读者，双重检查成例在 Java 语言里并不能成立，详情请见本章的附录。

15.2 单例模式的结构

单例模式有以下的特点：

- 单例类只能有一个实例。
- 单例类必须自己创建自己的惟一的实例。
- 单例类必须给所有其他对象提供这一实例。

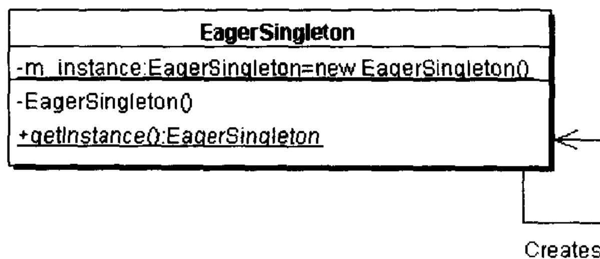


虽然单例模式中的单例类被限定只能有一个实例，但是单例模式和单例类可以很容易被推广到任意且有有限多个实例的情况，这时候称它为多例模式（Multiton Pattern）和多例类（Multiton Class），请见“专题：多例（Multiton）模式与多语言支持”一章。单例类的简略类图如右图所示。

由于 Java 语言的特点，使得单例模式在 Java 语言的实现上有自己的特点。这些特点主要表现在单例类如何将自己实例化上。

饿汉式单例类

饿汉式单例类是在 Java 语言里实现起来最为简便的单例类，下图所示的类图描述了一个饿汉式单例类的典型实现。



从图中可以看出，此类已经自己将自己实例化，其源代码如代码清单 1 所示。

代码清单 1：饿汉式单例类

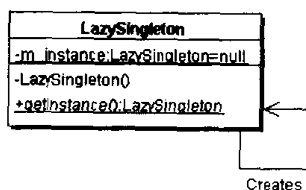
```
public class EagerSingleton
{
    private static final EagerSingleton m_instance =
        new EagerSingleton();
  /**
   *  私有的默认构造子
   */
    private EagerSingleton() { }
  /**
   *  静态工厂方法
   */
    public static EagerSingleton getInstance()
    {
        return m_instance;
    }
}
```

读者可以看出，在这个类被加载时，静态变量 m_instance 会被初始化，此时类的私有构造子会被调用。这时候，单例类的惟一实例就被创建出来了。

Java 语言中单例类的一个最重要的特点是类的构造子是私有的，从而避免外界利用构造子直接创建出任意多的实例。值得指出的是，由于构造子是私有的，因此此类不能被继承。

懒汉式单例类

与饿汉式单例类相同之处是，类的构造子是私有的。与饿汉式单例类不同的是，懒汉式单例类在第一次被引用时将自己实例化。如果加载器是静态的，那么在懒汉式单例类被加载时不会将自己实例化。如下图所示，类图中给出了一个典型的饿汉式单例类实现。



懒汉式单例类的源代码如代码清单 2 所示。

代码清单 2：懒汉式单例类

```
package com.javapatterns.singleton.demos;
public class LazySingleton
{
    private static LazySingleton
        m_instance = null;
   /**
    * 私有的默认构造子，保证外界无法直接实例化
    */
    private LazySingleton() { }
   /**
    * 静态工厂方法，返还此类的惟一实例
    */
    synchronized public static LazySingleton
        getInstance()
    {
        if (m_instance == null)
        {
            m_instance = new LazySingleton();
        }
        return m_instance;
    }
}
```

读者可能会注意到，在上面给出懒汉式单例类实现里对静态工厂方法使用了同步化，以处理多线程环境。有些设计师在这里建议使用所谓的“双重检查成例”。必须指出的是，“双重检查成例”不可以在 Java 语言中使用。不十分熟悉的读者，可以看看后面给出的内容。

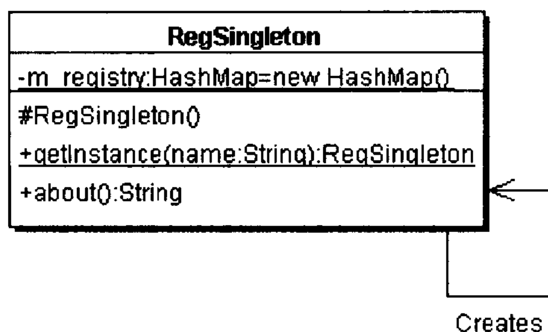
同样，由于构造子是私有的，因此，此类不能被继承。

饿汉式单例类在自己被加载时就将自己实例化。即便加载器是静态的，在饿汉式单例类被加载时仍会将自己实例化。单从资源利用效率角度来讲，这个比懒汉式单例类稍差些。从速度和反应时间角度来讲，则比懒汉式单例类稍好些。然而，懒汉式单例类在实例化时，必须处理好在多个线程同时首次引用此类时的访问限制问题，特别是当单例类作为资源控制器在实例化时必然涉及资源初始化，而资源初始化很有可能耗费时间。这意味着出现多线程同时首次引用此类的几率变得较大。

饿汉式单例类可以在 Java 语言内实现，但不易在 C++内实现，因为静态初始化在 C++里没有固定的顺序，因而静态的 m_instance 变量的初始化与类的加载顺序没有保证，可能会出问题。这就是为什么 GoF 在提出单例类的概念时，举的例子是懒汉式的。他们的书影响之大，以致 Java 语言中单例类的例子也大多是懒汉式的。实际上，本书认为饿汉式单例类更符合 Java 语言本身的特点。

登记式单例类

登记式单例类是 GoF 为了克服饿汉式单例类及懒汉式单例类均不可继承的缺点而设计的。本书把他们的例子翻译为 Java 语言，并将它自己实例化的方式从懒汉式改为饿汉式。只是它的子类实例化的方式只能是懒汉式的，这是无法改变的。如下图所示，是登记式单例类的一个例子，图中的关系线表明此类已将自己实例化。



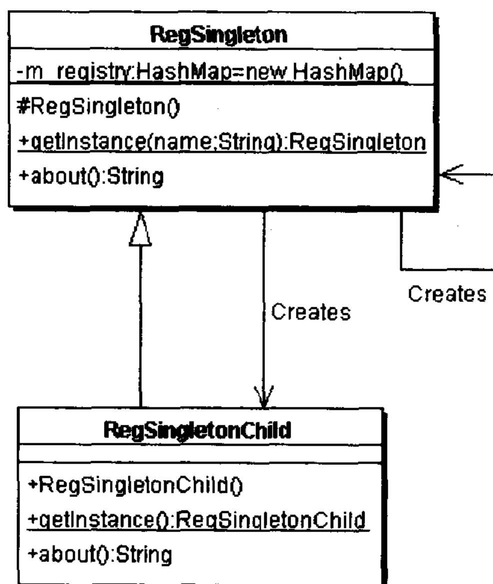
登记式单例类的源代码如代码清单 3 所示。

代码清单 3：登记式单例类的源代码

```
import java.util.HashMap;
public class RegSingleton
{
    static private HashMap m_registry = new HashMap();
    static
    {
        RegSingleton x = new RegSingleton();
        m_registry.put( x.getClass().getName() ,   x);
    }
   /**
    *  保护的默认构造子
    */
    protected RegSingleton() { }
   /**
    *  静态工厂方法，返还此类惟一的实例
    */
    static public RegSingleton getInstance(String name)
    {
        if (name == null)
        {
            name = "com.javapatterns.singleton.demos.RegSingleton";
        }
        if (m_registry.get(name) == null)
        {
            try
```

```
            {
                m_registry.put( name,
                    Class.forName(name).newInstance() ) ;
            }
            catch(Exception e)
            {
                System.out.println("Error happened.");
            }
        }
        return   (RegSingleton) (m_registry.get(name) );
    }
    /**
     *   一个示意性的商业方法
     */
    public String about()
    {
        return "Hello，  I am RegSingleton.";
    }
}
```

它的子类 RegSingletonChild 需要父类的帮助才能实例化。下图所示是登记式单例类子类的一个例子，图中的关系表明此类是由父类将子类实例化的。



其子类的源代码如代码清单 4 所示。

代码清单 4：登记式单例类的子类

```
import java.util.HashMap;
public class RegSingletonChild extends RegSingleton
{
    public RegSingletonChild() {}
    /**
```

```
     *  静态工厂方法
     */
    static public RegSingletonChild getInstance()
    {
        return (RegSingletonChild)
            RegSingleton.getInstance(
            "com.javapatterns.singleton.demos.RegSingletonChild" );
    }
   /**
    *  一个示意性的商业方法
    */
    public String about()
    {
        return "Hello， I am RegSingletonChild.";
    }
}
```

在 GoF 原始的例子中并没有 getInstance()方法，这样得到子类必须调用的 getInstance（String name）方法并传入子类的名字，因此很不方便。本章在登记式单例类子类的例子里，加入了 getInstance()方法，这样做的好处是 RegSingletonChild 可以通过这个方法返还自己的实例。而这样做的缺点是，由于数据类型不同，无法在 RegSingleton 提供这样一个方法。

由于子类必须允许父类以构造子调用产生实例，因此，它的构造子必须是公开的。这样一来，就等于允许了以这样方式产生实例而不在父类的登记中。这是登记式单例类的一个缺点。

GoF 曾指出，由于父类的实例必须存在才可能有子类的实例，这在有些情况下是一个浪费。这是登记式单例类的另一个缺点。

15.3 在什么情况下使用单例模式

使用单例模式的条件

使用单例模式有一个必要条件：在一个系统要求一个类只有一个实例时才应当使用单例模式。反过来说，如果一个类可以有几个实例共存，那么就没有必要使用单例类。

但是有经验的读者可能会看到很多不当地使用单例模式的例子，可见做到上面这一点并不容易，下面就是这样的一些情况。

例子一

问：我的一个系统需要一些“全程”变量。学习了单例模式后，我发现可以使用一个

单例类盛放所有的“全程”变量。请问这样做对吗？

答：这样做是违背单例模式的用意的。单例模式只应当在有真正的“单一实例”的需求时才可使用。

一个设计得当的系统不应当有所谓的“全程”变量，这些变量应当放到它们所描述的实体所对应的类中去。将这些变量从它们所描述的实体类中抽出来，放到一个不相干的单例类中去，使得这些变量产生错误的依赖关系和耦合关系。

例子二

问：我的一个系统需要管理与数据库的链接。学习了单例模式后，我发现可以使用一个单例类包装一个 Connection 对象，并在 finalize()方法中关闭这个 Connection 对象。这样的话，在这个单例类的实例没有被人引用时，这个 finalize()对象就会被调用，因此 Connection 对象就会被释放，这多妙啊。

答：这样做是不恰当的。除非有单一实例的需求，否则不要使用单例模式。在这里 Connection 对象可以同时有几个实例共存，不必是单一实例。

单例模式有很多的错误使用案例都与此例子相似，它们都是试图使用单例模式管理共享资源的生命周期，这是不恰当的。

15.4 单例类的状态

有状态的单例类

一个单例类可以是有状态的（stateful），一个有状态的单例对象一般也是可变（mutable）单例对象。

有状态的可变的单例对象常常当做状态库（repositary）使用。比如一个单例对象可以持有一个 int 类型的属性，用来给一个系统提供一个数值惟一的序列号码，作为某个贩卖系统的账单号码。

当然，一个单例类可以持有一个聚集，从而允许存储多个状态。

没有状态的单例类

另一方面，单例类也可以是没有状态的（stateless），仅用做提供工具性函数的对象。既然是为了提供工具性函数，也就没有必要创建多个实例，因此使用单例模式很合适。一个没有状态的单例类也就是不变（Immutable）单例类。关于不变模式，读者可以参见本书的“不变（Immutable）模式”一章。



多个 JVM 系统的分散式系统

EJB 容器有能力将一个 EJB 的实例跨过几个 JVM 调用。由于单例对象不是 EJB，因此，单例类局限于某一个 JVM 中。换言之，如果 EJB 在跨过 JVM 后仍然需要引用同一个单例类的话，这个单例类就会在数个 JVM 中被实例化，造成多个单例对象的实例出现。一个 J2EE 应用系统可能分布在数个 JVM 中，这时候不一定需要 EJB 就能造成多个单例类的实例出现在不同 JVM 中的情况。

如果这个单例类是没有状态的，那么就没有问题。因为没有状态的对象是没有区别的。但是如果这个单例类是有状态的，那么问题就来了。举例来说，如果一个单例对象可以持有一个 int 类型的属性，用来给一个系统提供一个数值惟一的序列号码，作为某个贩卖系统的账单号码的话，用户会看到同一个号码出现好几次。

在任何使用了 EJB、RMI 和 JINI 技术的分散式系统中，应当避免使用有状态的单例模式。

多个类加载器

同一个 JVM 中会有多个类加载器，当两个类加载器同时加载同一个类时，会出现两个实例。在很多 J2EE 服务器允许同一个服务器内有几个 Servlet 引擎时，每一个引擎都有独立的类加载器，经由不同的类加载器加载的对象之间是绝缘的。

比如一个 J2EE 系统所在的 J2EE 服务器中有两个 Servlet 引擎：一个作为内网给公司的网站管理人员使用；另一个给公司的外部客户使用。两者共享同一个数据库，两个系统都需要调用同一个单例类。如果这个单例类是有状态的单例类的话，那么内网和外网用户看到的单例对象的状态就会不同。

除非系统有协调机制，不然在这种情况下应当尽量避免使用有状态的单例类。

15.5　一个实用的例子：属性管理器

什么是属性文件

这里给出一个读取属性（properties）文件的单例类，作为单例模式的一个实用的例子。属性文件如同老式的视窗编程时的.ini 文件，用于存放系统的配置信息。配置信息在属性文件中以属性的方式存放，一个属性就是两个字符串组成的对子，其中一个字符串是键（key），另一个字符串是这个键的值（value）。

大多数的系统都有一些配置常量，这些常量如果是存储在程序内部的，那么每一次修改这些常量都需要重新编译程序。将这些常量放在配置文件中，系统通过访问这个配置文件取得配置常量，就可以通过修改这个配置文件而无需修改程序而达到更改系统配置的目

的。

系统也可以在配置文件中存储一些工作环境信息，这样在系统重启时，这些工作信息可以延续到下一个运行周期中。

假定需要读取的属性文件就在当前目录中，且文件名为 singleton.properties。这个文件中的一些属性项如代码清单 5 所示。

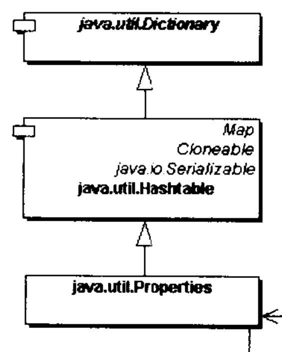
代码清单 5：属性文件内容

```
node1.item1=How
node1.item2=are
node2.item1=you
node2.item2=doing
node3.item1=?
```

例如，node1.item1 就是一个键，而 How 就是这个键所对应的值。

Java 属性类

Java 提供了一个工具类，称做属性类，可以用来完成 Java 属性和属性文件的操作。这个属性类的继承关系可以从右图所示的类图中看清楚。



属性类提供了读取属性和设置属性的各种方法。其中读取属性的方法有：

- contains(Object value)、containsKey(Object key)：如果给定的参数或属性关键字在属性表中有定义，该方法返回 True，否则返回 False。
- getProperty(String key)、getProperty(String key, String default)：根据给定的属性关键字获取关键字值。
- list(PrintStream s)、list(PrintWriter w)：在输出流中输出属性表内容。
- size()：返回当前属性表中定义的属性关键字个数。

设置属性的方法有：

- put(Object key, Object value)：向属性表中追加属性关键字和关键字的值。
- remove(Object key)：从属性表中删除关键字。

从属性文件加载属性的方法为 load(InputStream inStream)，可以从一个输入流中读入一个属性列，如果这个流是来自一个文件的话，这个方法就从文件中读入属性。

将属性存入属性文件的方法有几个，最重要的一个是 store(OutputStream out, String header)，将当前的属性列写入一个输出流，如果这个输出流是导向一个文件的，那么这个方法就将属性流存入文件。

为什么需要使用单例模式

属性是系统的一种“资源”，应当避免有多于一个的对象读取，特别是存储属性。此

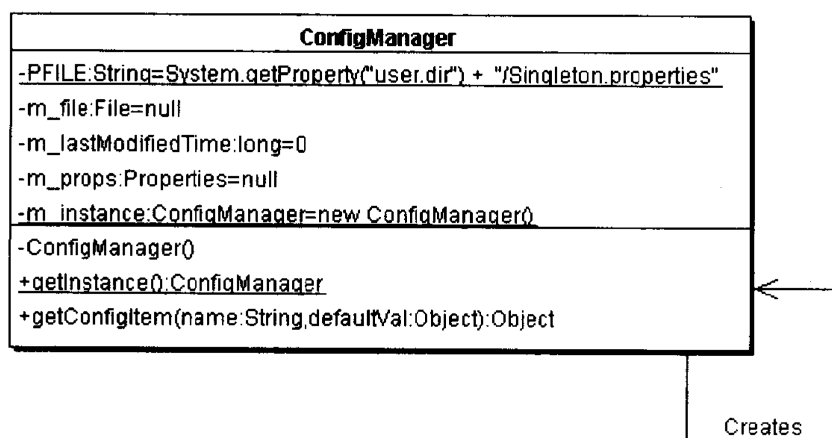
外，属性的读取可能会在很多地方发生，创建属性对象的地方应当在哪里不是很清楚。换言之，属性管理器应当自己创建自己的实例，并且自己向系统全程提供这一实例。

因此，属性文件管理器应当是由一个单例模式负责的。

系统设计

系统的核心是一个属性管理器，也就是一个叫做 ConfigManager 的类，这个类应当是一个单例类。因此，这个类应当有一个静态工厂方法，不妨叫做 getInstance()，用于提供自己的实例。

为简单起见，本书在这里采取“饿汉”方式实现 ConfigManager。例子的类图如下图所示。



本例子的源代码如代码清单 6 所示。

代码清单 6：ConfigManager 的源代码

```
import java.util.Properties;
import java.io.FileInputStream;
import java.io.File;
public class ConfigManager
{
    /**
     * 属性文件全名
     */
     private static final String PFILE =
            System.getProperty("user.dir")
            + File.Separator + "Singleton.properties";
    /**
     * 对应于属性文件的文件对象变量
     */
     private File m_file = null;
    /**
     * 属性文件的最后修改日期
```

```
    */
    private long m_lastModifiedTime = 0;
   /**
    * 属性文件所对应的属性对象变量
    */
    private Properties m_props = null;
   /**
    * 本类可能存在的惟一的一个实例
    */
    private static ConfigManager m_instance =
        new ConfigManager();
   /**
    * 私有的构造子， 用以保证外界无法直接实例化
    */
    private ConfigManager()
    {
        m_file = new File(PFILE);
        m_lastModifiedTime = m_file.lastModified();
        if(m_lastModifiedTime == 0)
        {
            System.err.println(PFILE +
                " file does not exist!");
        }
        m_props = new Properties();
        try
        {
            m_props.load(new FileInputStream(PFILE));
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
   /**
    * 静态工厂方法
    * @return 返还 ConfigManager 类的单一实例
    */
    synchronized public static ConfigManager
        getInstance()
    {
        return m_instance;
    }
   /**
    * 读取一个特定的属性项
    *
    * @param name  属性项的项名
```

```
* @param defaultVal  属性项的默认值
* @return  属性项的值（如此项存在），默认值（如此项不存在）
*/
final public Object getConfigItem(
    String name, Object defaultVal)
{
    long newTime = m_file.lastModified();
    // 检查属性文件是否被其他程序
    // （多数情况是程序员手动）修改过
    // 如果是，重新读取此文件
    if(newTime == 0)
    {
        // 属性文件不存在
        if(m_lastModifiedTime == 0)
        {
            System.err.println(PFILE
                + " file does not exist!");
        }
        else
        {
            System.err.println(PFILE
                + " file was deleted!!");
        }
        return defaultVal;
    }
    else if(newTime > m_lastModifiedTime)
    {
        // Get rid of the old properties
        m_props.clear();
        try
        {
            m_props.load(new FileInputStream(PFILE));
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    m_lastModifiedTime = newTime;
    Object val = m_props.getProperty(name);
    if( val == null )
    {
        return defaultVal;
    }
    else
    {
```

```
                return val;
            }
        }
    }
```

在上面直接使用了一个局域的常量存储属性文件的路径。在实际的系统中，读者可以采取更灵活的方式将属性文件的路径传入。

读者可以看到，这个管理器类有一个很有意思的功能，即在每一次调用时，检查属性文件是否已经被更新过。如果确实已经被更新过的话，管理器会自动重新加载属性文件，从而保证管理器的内容与属性文件的内容总是一致的。

怎样调用属性管理器

如代码清单 7 所示，其中演示了怎样调用 ConfigManager 来读取属性文件。

代码清单 7：调用 ConfigManager 类来读取属性文件

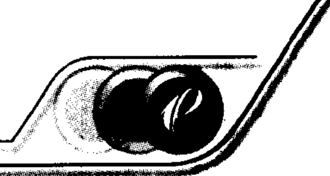
```
BufferedReader reader = new BufferedReader(
    new InputStreamReader(System.in));
System.out.println("Type quit to quit");
do
{
    System.out.print("Property item to read: ");
    String line = reader.readLine();
    if(line.equals("quit"))
    {
        break;
    }
    System.out.println(ConfigManager.getInstance()
        .getConfigItem(line,  "Not found."));
} while(true);
```

上面代码运行时的情况如下图所示。

```
 Total number of kept objects: 0
 Total number of kept objects: 1
Type quit to quit
Property item to read: node1.item1
How
Property item to read: node1.item2
are
Property item to read: node2.item1
you
Property item to read: node2.item2
doing
Property item to read: node3.item1
?
Property item to read: quit
 Total number of kept objects: 1
 Total number of kept objects: 0

E:\JavaPatterns\classes>
```

感兴趣的读者可以参考本书的"专题：XMLProperties 与适配器模式"一章，那里对使用 Java 属性类和 XML 文件格式做了有用的讨论。



15.6　Java 语言中的单例模式

Java 语言中就有很多单例模式的应用实例，这里讨论比较有名的几个。

Java 的 Runtime 对象

在 Java 语言内部，java.lang.Runtime 对象就是一个使用单例模式的例子。在每一个 Java 应用程序里面，都有惟一的一个 Runtime 对象。通过这个 Runtime 对象，应用程序可以与其运行环境发生相互作用。

Runtime 类提供一个静态工厂方法 getRuntime()::

```
public static Runtime getRuntime();
```

通过调用此方法，可以获得 Runtime 类惟一的一个实例：

```
Runtime rt = Runtime getRuntime();
```

Runtime 对象通常的用途包括：执行外部命令；返回现有内存即全部内存；运行垃圾收集器；加载动态库等。下面的例子演示了怎样使用 Runtime 对象运行一个外部程序，如代码清单 8 所示。

代码清单 8：使用 Runtime 对象运行一个外部命令

```
import java.io.*;
public class CmdTest
{
    public static void main(String[] args) throws IOException
    {
        Process proc = Runtime.getRuntime().exec("notepad.exe");
    }
}
```

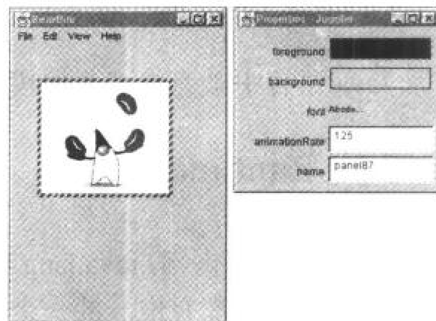
上面的程序在运行时会打开 notepad 程序。应当指出的是，在 Windows 2000 环境下，如果需要打开一个 Word 文件，而又不想指明 Word 软件安装的位置时，可以使用下面的做法：

```
Process proc = Runtime.getRuntime().exec(
    "cmd /E:ON /c start MyDocument.doc");
```

在上面，被执行的命令是 start MyDocument.doc，开关 E:ON 指定 DOS 命令处理器允许命令扩展，而开关/C 指明后面跟随的字符串是命令，并在执行命令后关闭 DOS 窗口，start 命令会开启一个单独的窗口执行所提供的命令。

Introspector 类

一般的应用程序可能永远也不会直接用到 Introspector 类，但读者应该知道 Introspector 是做什么的。Sun 提供了一个叫做 BeanBox 的系统，允许动态地加载 JavaBean，并动态地修改其性质。BeanBox 在运行时的情况如右图所示。



在上面的图中显示了 BeanBox 最重要的两个视窗，一个叫做 BeanBox 视窗，另一个叫做性质视窗。在上面的 BeanBox 视窗中显示了一个 Juggler Bean 被放置到视窗中的情况。相应地，在性质视窗中显示了 Juggler Bean 的所有性质。所有的 Java 集成环境都提供这种功能，这样的系统就叫做 BeanBox 系统。

BeanBox 系统使用一种自省（Introspection）过程来确定一个 Bean 所输出的性质、事件和方法。这个自省机制是通过自省者类，也即 java.util.Introspector 类实现的；这个机制是建立在 Java 反射（Reflection）机制和命名规范的基础之上的。比如，Introspector 类可以确定 Juggler Bean 所支持的所有的性质，这是因为 Introspector 类可以得到所有的方法，然后将其中的取值和赋值方法以及它们的特征加以比较，从而得出结果。

显然，在整个 BeanBox 系统中只需要一个 Introspector 对象，如下图所示，便是这个类的结构图。

java.beans.Introspector

```
+getBeanInfo:BeanInfo
+getBeanInfo:BeanInfo
+getBeanInfo:BeanInfo
+decapitalize:String
+getBeanInfoSearchPath:String[]
+setBeanInfoSearchPath:void
+flushCaches:void
+flushFromCaches:void
-Introspector
-getBeanInfo:GenericBeanInfo
-findInformant:BeanInfo
-getTargetPropertyInfo:PropertyDescriptor[]
addProperty:void
-getTargetEventInfo:EventSetDescriptor[]
addEvent:void
-getTargetMethodInfo:MethodDescriptor[]
-addMethod:void
-makeQualifiedMethodName:String
-getTargetDefaultEventIndex:int
-getTargetDefaultPropertyIndex:int
-getTargetBeanDescriptor:BeanDescriptor
-isEventHandler:boolean
-getPublicDeclaredMethods:Method[]
-internalFindMethod:Method
-internalFindMethod:Method
findMethod:Method
findMethod:Method
isSubclass:boolean
-throwsException:boolean
instantiate:Object
```

可以看出，Introspector 类的构造子是私有的，一个静态工厂方法 instantiate()提供了 Introspector 类的惟一实例。换言之，这个类是单例模式的应用。

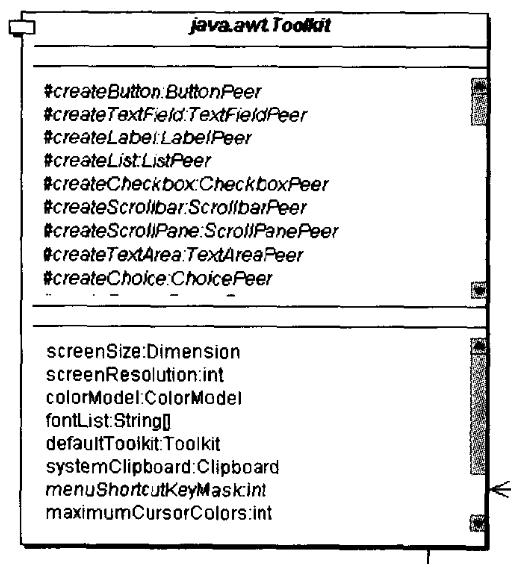
java.awt.Toolkit 类

Toolkit 类是一个非常有趣的单例模式的例子。Toolkit 使用单例模式创建所谓的 Toolkit 的默认对象，并且确保这个默认实例在整个系统中是惟一的。Toolkit 类提供了一个静态的方法 getDefaultToolkit()来提供这个惟一的实例，这个方法相当于懒汉式的单例方法，因此整个方法都是同步化的，如代码清单 9 所示。

代码清单 9：getDefaultToolkit()方法

```
public static synchronized Toolkit
      getDefaultToolkit()
{
      ......
}
```

Toolkit 类的类图如下图所示。



其中，性质 defaultToolkit 实际上就是静态的 getDefaultToolkit 类。

有趣的是，由于 Toolkit 是一个抽象类，因此如果其子类提供一个私有的构造子，那么其子类便是一个正常的单例类；而如果其子类作为具体实现提供一个公开的构造子，这时候这个具体子类便是“不完全”的单例类。关于“不完全”的单例类的讨论请见本章后面的“专题：不完全的单例类”一节。

模版方法模式

同时，熟悉模版方法模式的读者可以看出，getDefaultToolkit()方法实际上是一个模版方法。私有构造子是推迟到子类实现的剩余逻辑，根据子类对这个剩余逻辑的不同实现，子类就可以提供完全不同的行为。对 Toolkit 的子类而言，私有构造子依赖于操作系统，

不同的子类可以根据不同的操作系统而给出不同的逻辑，从而使 Toolkit 的子类对不同的操作系统给出不同的行为。

javax.swing.TimerQueue 类

这是一个不完全的单例类，由于这个类是在 Swing 的定时器类中使用的，因此将在“观察者模式与 Swing 定时器”一章中介绍。

15.7 专题：不完全的单例类

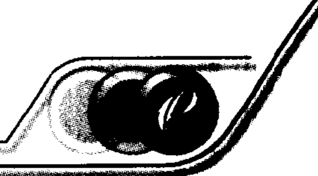
什么是不完全的单例类

估计有些读者见过下面这样的“不完全”的单例类，如代码清单 10 所示。

代码清单 10：“不完全”单例类

```
package com.javapatterns.singleton.demos;
public class LazySingleton
{
    private static LazySingleton
        m_instance = null;
   /**
    * 公开的构造子，外界可以直接实例化
    */
    public LazySingleton() { }
   /**
    * 静态工厂方法
    * @return 返还 LazySingleton 类的惟一实例
    */
    synchronized public static
        LazySingleton getInstance()
    {
        if (m_instance == null)
        {
            m_instance = new LazySingleton();
        }
        return m_instance;
    }
}
```

上面的代码乍看起来是一个“懒汉”式单例类，仔细一看，发现有一个公开的构造子。由于外界可以使用构造子创建出任意多个此类的实例，这违背了单例类只能有一个（或有限个）实例的特性，因此这个类不是完全的单例类。这种情况有时会出现，比如



javax.swing.TimerQueue 便是一例，关于这个类，请参见“观察者模式与 Swing 定时器”一章。

造成这种情况出现的原因有以下几种可能：

（1）初学者的错误。许多初学者没有认识到单例类的构造子不能是公开的，因此犯下这个错误。有些初学 Java 语言的学员甚至不知道一个 Java 类的构造子可以不是公开的。在这种情况下，设计师可能会通过自我约束，也就是说不去调用构造子的办法，将这个不完全的单例类在使用中作为一个单例类使用。

在这种情况下，一个简单的矫正办法，就是将公开的构造子改为私有的构造子。

（2）当初出于考虑不周，将一个类设计成为单例类，后来发现此类应当有多于一个的实例。为了弥补错误，干脆将构造子改为公开的，以便在需要多于一个的实例时，可以随时调用构造子创建新的实例。

要纠正这种情况较为困难，必须根据具体情况做出改进的决定。如果一个类在最初被设计成为单例类，但后来发现实际上此类应当有有限多个实例，这时候应当考虑是否将单例类改为多例类（Multiton）。

（3）设计师的 Java 知识很好，而且也知道单例模式的正确使用方法，但是，还是有意使用这种不完全的单例模式，因为他意在使用一种“改良”的单例模式。这时候，除去共有的构造子不符合单例模式的要求之外，这个类必须是很好的单例模式。

默认实例模式

有些设计师将这种不完全的单例模式叫做“默认实例模式”（Default Instance Pattern）。在所谓的“默认实例模式”里面，一个类提供静态的方法，如同单例模式一样，同时又提供一个公开的构造子，如同普通的类一样。

这样做的惟一好处是，这种模式允许客户端选择如何将类实例化：创建新的自己独有的实例，或者使用共享的实例。

这样一来，由于没有任何的强制性措施，客户端的选择不一定是合理的选择。其结果是设计师往往不会花费时间在如何提供最好的选择上，而是不恰当地将这种选择交给客户端的程序员，这样必然会导致不理想的设计和欠考虑的实现。

本书建议读者不要这样做。

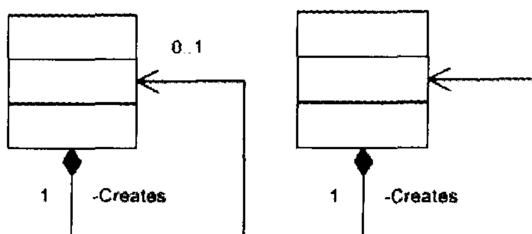
15.8 相关模式

有一些模式可以使用单例模式，如抽象工厂模式可以使用单例模式，将具体工厂类设计成单例类；建造模式可以使用单例模式，将具体建造类设计成单例类。

多例（Multiton）模式

正如同本章所说的，单例模式的精神可以推广到多于一个实例的情况。这时候这种类

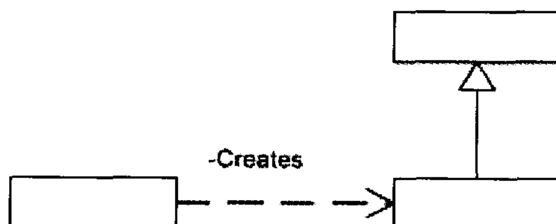
叫做多例类，这种模式叫做多例模式。单例类（左）和多例类（右）的类图如下图所示。



关于多例模式，请见本书“专题：多例（Multiton）模式与多语言支持”一章。

简单工厂（Simple Factory）模式

单例模式使用了简单工厂模式（又称为静态工厂方法模式）来提供自己的实例。在上面 ConfigManager 例子的代码中，静态工厂方法 getInstance()就是静态工厂方法。在 java.awt.Toolkit 类中，getDefaultToolkit()方法就是静态工厂方法。简单工厂模式的简略类图如下图所示。



本章讨论了单例模式的结构和实现方法。

单例模式是一个看上去很简单的模式，很多设计师最先学会的往往是单例模式。然而，随着 Java 系统日益变得复杂化和分散化，单例模式的使用变得比过去困难。

本书提醒读者在分散式的 Java 系统中使用单例模式时，尽量不要使用有状态的。

问答题

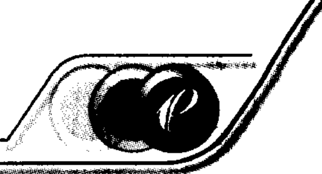
1. 为什么不使用一个静态的“全程”原始变量，而要建一个类呢？一个静态的原始变量当然只能有一个值，自然而然不就是“单例”的吗？

2. 举例说明如何调用 EagerSingleton 类。

3. 举例说明如何调用 RegSingleton 类和 RegSingletonChild 类。

4. 请问 java.lang.Math 类和 java.lang.StrictMath 类是否是单例模式？

5. 我们公司只购买了一个 JDBC 驱动软件的单用户使用许可，可否使用单例模式管理通过 JDBC 驱动软件连接的数据库？



问答题答案

1. 单例模式可以提供很复杂的逻辑，而一个原始变量不能自己初始化，不可能有继承的关系，没有内部结构。因此单例模式有很多优越之处。

在 Java 语言里并没有真正的“全程”变量，一个变量必须属于某一个类或者某一个实例。而在复杂的程序当中，一个静态变量的初始化发生在哪里常常是一个不易确定的问题。

当然，使用“全程”原始变量并没有什么错误，就好像选择使用 Fortran 语言而非 Java 语言编程并不是一种对错的问题一样。

2. 几种单例类的使用方法如代码清单 11 所示。

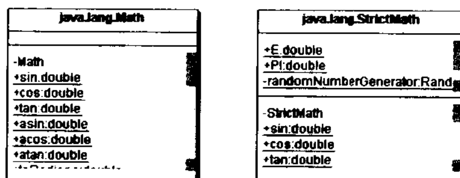
代码清单 11：几种单例类的使用方法

```
public class RegSingletonTest
{
    public static void main(String[] args)
    {
        //(1) Test eager
        System.out.println( EagerSingleton.getInstance());
        //(2) Test reg
        System.out.println(
            RegSingleton.getInstance(
            "com.javapatterns.singleton.demos.RegSingleton").about());
        System.out.println( RegSingleton.getInstance(null).about() );
        System.out.println(
            RegSingleton.getInstance(
            "com.javapatterns.singleton.demos.RegSingletonChild").about());
        System.out.println( RegSingletonChild.getInstance().about());
    }
}
```

3. 见上题答案。

4. 它们都不是单例类，原因如下。

这两个类均有一个私有的构造子。但是这仅仅是单例模式的必要条件，而不是充分条件。回顾本章开始提出的单例模式的三个特性，可以看出无论是 Math 还是 StrictMath 都没有为外界提供任何自身的实例。实际上，这两个类都是被设计来提供静态工厂方法和常量的，因此从来就不需要它们的实例，这才是它们的构造子是私有的原因。Math 和 StrictMath 类的类图如下图所示。



5. 这样做是可行的，只是必须注意当使用在分散式系统中的时候，不一定能保证单例类实例的惟一性。

15.9 附录：双重检查成例的研究

成例是一种代码层次上的模式，是在比设计模式的层次更具体的层次上的代码技巧。成例往往与编程语言密切相关。双重检查成例（Double Check Idiom）是从 C 语言移植过来的一种代码模式。在 C 语言里，双重检查成例常常用在多线程环境中类的晚实例化（Late Instantiation）里。

本节之所以要介绍这个成例（严格来讲，是介绍为什么这个成例不成立），是因为有很多人认为双重检查成例可以使用在“懒汉”单例模式里面。

什么是双重检查成例

为了解释什么是双重检查成例，请首先看看下面没有使用任何线程安全考虑的错误例子。

从单线程的程序谈起

首先考虑一个单线程的版本，如代码清单 12 所示。

代码清单 12：没有使用任何线程安全措施的一个例子

```
// Single threaded version
class Foo
{
    private Helper helper = null;
    public Helper getHelper()
    {
        if (helper == null)
        {
            helper = new Helper();
        }
        return helper;
    }
    // other functions and members...
}
```

这是一个错误的例子，详情请见下面的说明。

写出这样的代码，本意显然是要保持在整个 JVM 中只有一个 Helper 的实例。因此，才会有 if (helper == null) 的检查。非常明显的是，如果在多线程的环境中运行，上面的代码会有两个甚至两个以上的 Helper 对象被创建出来，从而造成错误。

但是，想像一下在多线程环境中的情形就会发现，如果有两个线程 A 和 B 几乎同时到达 if (helper == null)语句的外面的话，假设线程 A 比线程 B 早一点点，那么：



（1）A 会首先进入 if (helper == null)块的内部，并开始执行 new Helper()语句。此时，helper 变量仍然是 null，直到线程 A 的 new Helper()语句返回并给 helper 变量赋值为止。

（2）但是，线程 B 并不会在 if (helper == null)语句的外面等待，因为此时 helper == null 是成立的，它会马上进入 if (helper == null)语句块的内部。这样，线程 B 会不可避免地执行 helper = new Helper();语句，从而创建出第二个实例来。

（3）线程 A 的 helper = new Helper();语句执行完毕后，helper 变量得到了真实的对象引用，(helper == null)不再为真。第三个线程不会再进入 if (helper == null)语句块的内部了。

（4）线程 B 的 helper = new Helper();语句也执行完毕后，helper 变量的值被覆盖。但是第一个 Helper 对象被线程 A 引用的事实不会改变。

这时，线程 A 和 B 各自拥有一个独立的 Helper 对象，而这是错误的。

线程安全的版本

为了克服没有线程安全的缺点，下面给出一个线程安全的例子，如代码清单 13 所示。

代码清单 13：这是一个正确的答案

```
// Correct multithreaded version
class Foo
{
    private Helper helper = null;
    public synchronized Helper getHelper()
    {
        if (helper == null)
        {
            helper = new Helper();
            return helper;
        }
    }
    // other functions and members...
}
```

显然，由于整个静态工厂方法都是同步化的，因此，不会有两个线程同时进入这个方法。因此，当线程 A 和 B 作为第一批调用者同时或几乎同时调用此方法时：

（1）早到一点的线程 A 会率先进入此方法，同时线程 B 会在方法外部等待。

（2）对线程 A 来说，helper 变量的值是 null，因此 helper = new Helper(); 语句会被执行。

（3）线程 A 结束对方法的执行，helper 变量的值不再是 null。

（4）线程 B 进入此方法，helper 变量的值不再是 null，因此 helper = new Helper(); 语句不会被执行。线程 B 取到的是 helper 变量所含有的引用，也就是对线程 A 所创立的 Helper 实例的引用。

显然，线程 A 和 B 持有同一个 Helper 实例，这是正确的。

画蛇添足的“双重检查”

但是，仔细审查上面的正确答案会发现，同步化实际上只在 helper 变量第一次被赋值

之前才有用。在 helper 变量有了值以后，同步化实际上变成了一个不必要的瓶颈。如果能有一个方法去掉这个小小的额外开销，不是更加完美了吗？因此，就有了下面这个设计"巧妙"的双重检查成例。在读者向下继续读之前，有必要提醒一句：正如本小节的标题所标明的那样，这是一个反面教材，因为双重检查成例在 Java 编译器里无法实现，如代码清单 14 所示。

代码清单 14：使用双重检查成例的懒汉式单例模式

```
// Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo
{
    private Helper helper = null;
    public Helper getHelper()
    {
        if (helper == null)      //第一次检查(位置 1)
        {
            //这里会有多于一个的线程同时到达 (位置 2)
            synchronized(this)
            {
                //这里在每个时刻只能有一个线程 (位置 3)
                if (helper == null) //第二次检查 (位置 4)
                {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
    // other functions and members...
}
```



这是一个错误的例子，详情请见下面的解释。

对于初次接触双重检查成例的读者来说，这个技巧的思路并不明显易懂。因此，本节在这里给出一个详尽的解释。同样，这里假设线程 A 和 B 作为第一批调用者同时或几乎同时调用静态工厂方法。

（1）因为线程 A 和 B 是第一批调用者，因此，当它们进入此静态工厂方法时，helper 变量是 null。因此，线程 A 和 B 会同时或几乎同时到达位置 1。

（2）假设线程 A 会首先到达位置 2，并进入 synchronized(this)到达位置 3。这时，由于 synchronized(this) 的同步化限制，线程 B 无法到达位置 3，而只能在位置 2 等候。

（3）线程 A 执行 helper = new Helper()语句，使得 helper 变量得到一个值，即对一个 Helper 对象的引用。此时，线程 B 只能继续在位置 2 等候。

（4）线程 A 退出 synchronized(this)，返回 Helper 对象，退出静态工厂方法。

（5）线程 B 进入 synchronized(this)块，达到位置 3，进而达到位置 4。由于 helper 变量已经不是 null 了，因此线程 B 退出 synchronized(this)，返回 helper 所引用的 Helper 对象(也就是线程 A 所创建的 Helper 对象)，退出静态工厂方法。

到此为止，线程 A 和线程 B 得到了同一个 Helper 对象。可以看到，在上面的方法 getInstance()中，同步化仅用来避免多个线程同时初始化这个类，而不是同时调用这个静态工厂方法。如果这是正确的，那么使用这一个成例之后，“懒汉式”单例类就可以摆脱掉同步化瓶颈，达到一个很妙的境界，如代码清单 15 所示。

代码清单 15：使用了双重检查成例的懒汉式单例类

```
public class LazySingleton
{
    private static LazySingleton m_instance = null;
    private LazySingleton() { }
   /**
    *  静态工厂方法
    */
    public static LazySingleton getInstance()
    {
        if (m_instance == null)
        {
            //More than one threads might be here!!!
            synchronized(LazySingleton.class)
            {
                if (m_instance == null)
                {
                    m_instance = new LazySingleton();
                }
            }
        }
        return m_instance;
    }
}
```



这是一个错误的例子，请见下面的解释。

第一次接触到这个技巧的读者必定会有很多问题，诸如第一次检查或者第二次检查可不可以省掉等。回答是：按照多线程的原理和双重检查成例的预想方案，它们是不可以省掉的。本节不打算讲解的原因在于双重检查成例在 Java 编译器中根本不能成立。

双重检查成例对 Java 语言编译器不成立

令人吃惊的是，在 C 语言里得到普遍应用的双重检查成例在多数的 Java 语言编译器

里面并不成立[BLOCH01, GOETZ01, DCL01]。上面使用了双重检查成例的“懒汉式”单例类，不能工作的基本原因在于，在 Java 编译器中，LazySingleton 类的初始化与 m_instance 变量赋值的顺序不可预料。如果一个线程在没有同步化的条件下读取 m_instance 引用，并调用这个对象的方法的话，可能会发现对象的初始化过程尚未完成，从而造成崩溃。

文献[BLOCH01]指出：一般而言，双重检查成立对 Java 语言来说是不成立的。

15.10 给读者的一点建议

有很多非常聪明的人在这个成例的 Java 版本上花费了非常多的时间，到现在为止人们得出的结论是：一般而言，双重检查成例无法在现有的 Java 语言编译器里工作[BLOCH01, GOETZ01, DCL01]。

读者可能会问，是否有可能通过某种技巧对上面的双重检查的实现代码加以修改，从而使某种形式的双重检查成例能在 Java 编译器下工作呢？这种可能性当然不能排除，但是除非读者对此有特别的兴趣，建议不要在这上面花费太多的时间。

在一般情况下，使用饿汉式单例模式或者对整个静态工厂方法同步化的懒汉式单例模式足以解决在实际设计工作中遇到的问题。

参考文献

[BLOCH01] Joshua Bloch. Effective Java–Programming Language Guide. published by Addison-Wesley, 2001

[NOBLE97] James Noble. GOF Patterns for GUI Design. preprint of Macquarie University, Sydney Australia, June 1, 1997

[DCL01] David Baconetal. The "Double-Checked Locking is Broken" Declaration. Preprint of University of Maryland, http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleChecked-Locking.html

[GOETZ01] Brian Goetz. Can ThreadLocal solve the double-checked locking problem. JavaWorld, November 2001, http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-dcl_p.html

[FOX01] Joshua Fox. When is a Singleton Not a Singleton. JavaWorld, January 2001

第 16 章　专题：单例模式与 MX 记录

本章的内容来自一个真实的华尔街金融网站项目。

本章假设读者对使用 JavaMail 库通过 SMTP（Simple Mail Transfer Protocol）服务器发送电子邮件有所了解，并且已经阅读过本书的“单例（Singleton）模式”一章。在运行本章提供的代码之前，请到 www.javasoft.com 下载最新版的 JavaMail 库。

16.1　问题与解决方案

问题

相信很多读者都接触过可以自动发送电子邮件的系统，大家想必知道可以利用 JavaMail 库通过一个 SMTP 服务器发送电子邮件的工作。一般来说，一个公司会有数个 SMTP 服务器，虽然每一台服务器都会定期停机维护检查，但是公司总会保持至少一台 SMTP 服务器正常运行，以便处理发送电子邮件的工作。

假设一个公司有一个 J2EE 系统需要自动发送电子邮件，而且这项工作必须是 24/7 运行的，也就是说是全天候的。换言之，不能因为某一台 SMTP 服务器停机就停止发送。与此同时，SMTP 服务器又不能不定时停机维护，这时候就需要系统架构设计师解决这个问题。

于是项目经理将几个设计师请到会议室，并说明问题所在，并要求设计师提供可能的设计方案。经过短暂的讨论，设计师们提出了两个可选方案。

第一个方案

安装一台自己的 SMTP 服务器，由系统维护人员维护这台服务器。这台服务器不一定要直接将电子邮件投递到客户那里，它只需要将电子邮件转投给公司里正在运行的 SMTP 服务器就可以了。由于 SMTP 服务器可以从 DNS 服务器那里拿到正在工作的 SMTP 服务器的名字，因此可以保证邮件总可以发送出去。

这当然是可以的，只是维护一台 SMTP 服务器等于添加了额外的维护工作，而且谁能保证这台服务器不需要停机检修呢？

第二个方案

在方案一中需要自己维护的那个 SMTP 服务器（以下称为源服务器）之所以能够找到

正在运行的 SMTP 服务器（以下称为目标服务器），并将邮件转投给它，是因为源服务器能够利用 DNS 的目录服务进行目录查询。DNS 会向整个网络提供所有登记过的 SMTP 服务器的名字，这样源服务器可以一个一个地试验目标服务器清单上的所有名字，直到找到一台正常运行的目标服务器为止。

使用 Java 的 JNDI 功能，Java 程序可以做同样的事情。这也就是说，可以写一个 Java 程序自动从 DNS 服务器那里得到一个公司内登记过的所有的 SMTP 服务器的清单（称为 MX 记录），然后让它一个个地试验清单上所有的服务器，直到把邮件送出去为止。由于公司内总会有至少一台 SMTP 服务器是正常工作的，这样就可以保证总可以将邮件送出去。

第二个方案的进一步完善

如果只需要多写 20 行代码，就可以省去日复一日的服务器维护工作，这当然是再好不过的。于是，这个方案被列为首选方案加以讨论，以便能够最终得到一个完善的解决方案。

下面就是综合讨论中提出的各种意见达成的设计方案：

（1）既然需要 JNDI 和 DNS，那么不妨到 Sun Microsystem 的网站去下载最新的库。

（2）不应当每一次发送邮件时都做 DNS 查询，应当创建一个类负责查询和保存查询所得的结果。

（3）MX 记录是一个相当静态的表，所以整个系统只需要一份表。系统在任何时候需要发送电子邮件，只需向这个对象调用这份列表即可。

所有这些都指向单例模式。

单例模式的使用

不能忘记本书引入这个例子是为了说明单例模式的使用，因此，有必要借此机会强调一下为什么这里应当使用单例模式，以及在这里使用单例模式应当注意的地方。

首先，正如设计师们在讨论中所指出的，在整个系统中只需要一份这样的 MX 记录表。这当然就是使用单例模式的最重要的理由。

其次，发送电子邮件的操作可能随时触发。也就是说，这个保存 MX 记录表的对象应当在全运行时期存在，而不应当被垃圾收集器所收集。了解垃圾收集器的读者可能会意识到，这就需要系统内保持至少一个对它的引用。那么由谁来保存对这个对象的引用，以保证它不被收集呢？当然是它自己最好。这样，只要这个单例对象一旦被创建，就会永远不被收集，直到服务器环境被重启为止。这也就是在这里使用单例模式的另一个重要的理由。

因此，单例模式最适合使用在这个系统设计中。

为了给不熟悉 JNDI、DNS 和 MX 记录等概念的读者一个熟悉这些概念的机会，在进行系统设计之前对这些概念进行一下复习。已经熟悉这些概念的读者可以跳过下面的几个小节，直接阅读“系统设计”一节。



16.2　目录服务与 MX 记录

命名 - 目录服务

一个生活中的命名 - 目录服务的例子就是电话台提供的电话目录查询服务。这个服务将把列在电话簿上的商家名字、地址和电话联系起来。

计算机的命名 - 目录服务是计算机系统的一个基本工具，而且比电话目录服务更加强大。命名 - 目录服务将一个对象与一个名字联系起来，使得客户可以通过对象的名字找到这个对象。目录服务允许对象有属性，这样客户端通过查询找到对象后，可以进一步得到对象的属性，或者反过来根据属性查询对象。

最常见的目录服务包括 LDAP（Lightweight Directory Access Protocol）和 DNS（Domain Name Service）。

什么是 DNS

DNS 即域名服务（Domain Name Service），电脑用户在网络上找到其他的电脑用户必须通过域名服务。在国际网络以及任何一个建立在 TCP/IP 基础之上的网络上，每一台电脑都有一个惟一的 IP 地址（Internet Protocol Address）。这些 IP 地址就像街道上的门牌号码，使得其他的电脑能找到某一台电脑。DNS 服务器提供 DNS 服务。

MX 记录（MX Record）

MX（Mail Exchange）记录就是邮件交换记录。电子邮件服务器记录指定一台服务器接收某个域名的邮件。也就是说，发往 jeffcorp.com 的邮件将发往 mail.jeffcorp.com 的服务器，完成此项任务的 MX 记录应当像下面这样：

```
jeffcorp.com. IN MX 10 mail.jeffcorp.com
```

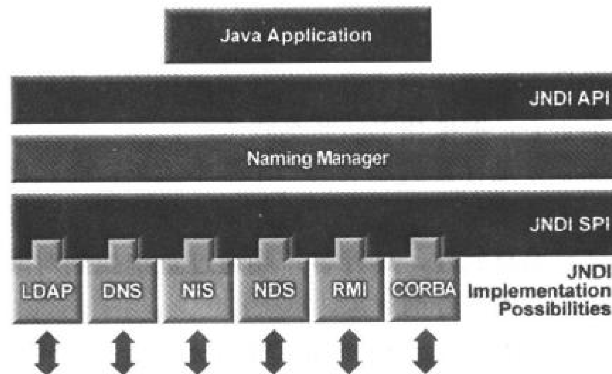
在上面的这个记录的最左边是国际网络上使用的电子邮件域名，第三列是一个数字 10，它代表此服务器的优先权是 10。通常来说，一个大型的系统会有数台电子邮件服务器，这些服务器可以依照优先权作为候补服务器使用。优先权必须是一个正整数，这个数字越低，表明优先权越高。

16.3　JNDI 架构介绍

JNDI 的全称是 Java 命名和地址界面（Java Naming and Directory Interface），是在 1997 年初由 Sun Microsystem 引进的，其目的是为 Java 系统提供支持各种目录类型的一个一般

性的访问界面。

JNDI 架构由 JNDI API 和 JNDI SPI 组成。JNDI API 使得一个 Java 应用程序可以使用一系列的命名（naming）和目录（directory）服务。JNDI SPI 是为服务提供商，包括目录服务提供商准备的，它可以让各种命名和目录服务能够以对应用程序透明的方式插入到系统里。在 JNDI 架构图中，给出了几个命名和目录服务作为例子，如下图所示。



JNDI API 由下面的四个库组成：

- javax.naming：包括了使用命名服务的类和接口。
- javax.naming.directory：扩展了核心库 javax.naming 的功能，提供目录访问功能。
- javax.naming.event：包括了命名和目录服务中所需要的事件通知机制的类和接口。
- javax.naming.ldap：包括了命名和目录服务中支持 LDAP（v3）所需要的类和接口。

构成了 JNDI SPI 的库为 javax.naming.spi 包括的类和接口允许各种的命名和目录服务提供商能将自己的软件服务构件加入到 JNDI 架构中去。

在 JNDI 里，所有的命名和目录操作都是相对于某一个 context（环境）而言的，JNDI 并没有任何的绝对根目录。JNDI 定义一个初始环境对象，称为 InitialContext，来提供命名和目录操作的起始点。一旦得到了初始环境，就可以使用初始环境查询其他环境和对象。

16.4 如何使用 JNDI 编程

本节提供一个非常简洁的介绍，讲解如何在 Java 中调用 JNDI 的功能。

建立开发环境

首先，如果读者没有最新版的 JNDI 和 DNS 库的话，请到 Sun Microsystem 的网站下载 JNDI 1.2.1 版和 DNS 1.2 版，然后将以下的三个 jar 文件放到 classpath 上面：

- jndi.jar
- dns.jar

- providerutil.jar

本章的代码需要以上三个包才能正常运行。

建立 Java 源文件

现在请读者建立一个 MXTest.java 文件，并加入以下的内容，如代码清单 1 所示。

代码清单 1：需要导入的类

```
import java.util.Hashtable;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;
import javax.naming.NamingException;
```

这会把下面编程所需要的类导入进来。本程序大部分的代码都在本程序的 main()方法中。

创建环境对象

在创建初始环境之前，必须指明两个环境性质参数，如代码清单 2 所示。第一个环境参数是服务提供商名，即 java.naming.factory.initial，这个性质的值应当是 com.sun.jndi.dns.DnsContextFactory，也就是 DNS 服务类的名字。有了这个类名，程序就可以动态地加载这个驱动类。第二个性质参数就是 DNS 服务器的 URL，即 java.naming.provider.url，这个性质的值应当是所在网络的一个合法 DNS 服务器的名字，对于本书作者来说就是 dns://dns01390.ny.jeffcorp.com。

代码清单 2：需要指明的系统性质参数

```
Hashtable env = new Hashtable();

env.put("java.naming.factory.initial",
        "com.sun.jndi.dns.DnsContextFactory");
env.put( "java.naming.provider.url",
        "dns://dns01390.ny.jeffcorp.com");
```

这时就可以创建初始环境了，如代码清单 3 所示。

代码清单 3：创建环境对象

```
DirContext dirContext = new InitialDirContext(env);
```

读取环境对象的属性

有了环境对象，就可以进行目录的属性查询了。一个环境对象的 getAttributes()方法

会返还这个环境对象的属性。由于这里需要的是某个域的所有 MX 记录，因此，可以使用下面的语句得到这些属性，如代码清单 4 所示。

代码清单 4：读取环境对象的属性

```
Attributes attrs = dirContext.getAttributes(
    "jeffcorp.com", new String[]{"MX"});
```

在得到这些属性后，可以通过属性对象的遍历方法读出每一条 MX 记录。

完整的程序

这个程序的完整源代码如代码清单 5 所示。

代码清单 5：MXTest 类的源代码

```
import java.util.Hashtable;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;
import javax.naming.NamingException;
public class MXTest
{
    public static void main(String[] args)
        throws NamingException
    {
        Hashtable env = new Hashtable();
        env.put("java.naming.factory.initial",
            "com.sun.jndi.dns.DnsContextFactory");
        env.put( "java.naming.provider.url",
             "dns://dns01390.ny.jeffcorp.com");
        // 创建环境对象
        DirContext dirContext = new
            InitialDirContext(env);
        // 读取环境对象的属性
        Attributes attrs = dirContext.getAttributes(
            "jeffcorp.com", new String[]{"MX"});
        for(NamingEnumeration ae = attrs.getAll();
            ae != null && ae.hasMoreElements();)
        {
            Attribute attr = (Attribute)ae.next();
            NamingEnumeration e = attr.getAll();
            while(e.hasMoreElements())
            {
                String element = e.nextElement().toString();
                System.out.println(element);
```

```
            }
        }
    }
}
```

运行结果

在运行程序后，打印出一列 SMTP 服务器的名字及优先权。下面就是一个典型的运行结果，如代码清单 6 所示。

代码清单 6：系统的运行结果

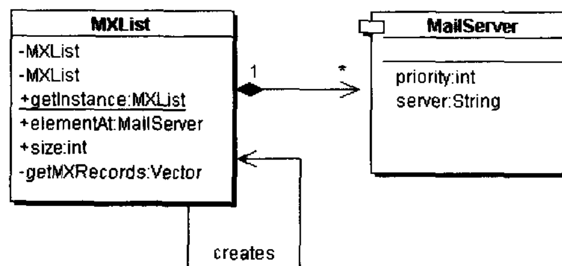
```
10 bird2–a.jeffcorp.com
10 bird2–b.jeffcorp.com
10 horse1–a.jeffcorp.com
10 horse1–b.jeffcorp.com
10 horse2–a.jeffcorp.com
10 horse2–b.jeffcorp.com
10 bird1–a.jeffcorp.com
10 bird1–b.jeffcorp.com
```

16.5 系 统 设 计

现在可以进行系统设计了。首先，系统需要一个单例类负责查询和保存列表。当然由于 SMTP 服务器的信息包括优先权和服务器名字，因此还需要一个 MailServer 类，用于存储关于一个 SMTP 服务器的信息。这样就有了下面介绍的两个类。

系统的静态结构

MX 信息查询系统的设计图如下图所示。



源代码

下面就是 MailServer 类的源代码，如代码清单 7 所示。这个类用于存储 SMTP 服务器的信息，包括服务器的名字和优先权。

代码清单 7：MailServer 类的源代码

```
package com.javapatterns.singleton.mxrecord;
public class MailServer
{
    private int priority;
    private String server;
    /**
     * 优先权的赋值方法
     */
    public void setPriority(int priority)
    {
        this.priority = priority;
    }
    /**
     * 服务器名的赋值方法
     */
    public void setServer(String server)
    {
        this.server = server;
    }
    /**
     * 优先权的取值方法
     */
    public int getPriority()
    {
        return priority;
    }
    /**
     * 服务器名的取值方法
     */
    public String getServer()
    {
        return server;
    }
}
```

下面就是系统的核心类 MXList 的源代码，如代码清单 8 所示。可以看到，这个类的构造子是私有的，因此，外界不能够直接将此类实例化。但是，这个类提供了一个静态工厂方法 getInstance()以提供自己的实例。

由于这个工厂方法实际上仅会返还惟一的一个实例，因此这个类是一个单例类。熟悉单例模式的读者可以发现，这个单例类采用的是懒汉式初始化方法。关于单例类及其实现方法，请参见本书的“单例（Singleton）模式”一章。

代码清单 8：MXList 类的源代码

```
package com.javapatterns.singleton.mxrecord;
```

```
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import java.util.StringTokenizer;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;
public class MXList
{
    private static MXList mxl = null;
    private Vector list = null;
    private static final String FACTORY_ENTRY =
        "java.naming.factory.initial";
    private static final String FACTORY_CLASS =
        "com.sun.jndi.dns.DnsContextFactory";
    private static final String PROVIDER_ENTRY =
        "java.naming.provider.url";
    private static final String MX_TYPE = "MX";
    private String dnsUrl = null;
    private String domainName = null;
  /**
   * 默认构造子是私有的，
   * 保证本类不能被外部直接实例化
   */
    private MXList() {}
  /**
   * 构造子是私有的，
   * 保证本类不能被外部直接实例化
   */
   private MXList(String dnsUrl,
        String domainName) throws Exception
   {
        this.dnsUrl = dnsUrl;
        this.domainName = domainName;

        this.list = getMXRecords(dnsUrl, domainName);
   }
  /**
   * 静态工厂方法，提供本类惟一的实例
   */
  public static synchronized MXList getInstance(
        String dnsUrl,
        String domainName) throws Exception
  {
```

```
        if (mxl == null)
        {
            mxl = new MXList(dnsUrl, domainName);
        }
        return mxl;
    }
    /**
     * 聚集方法，提供聚集元素
     */
    public MailServer elementAt(int index)
        throws Exception
    {
        return (MailServer) list.elementAt(index);
    }
    /**
     * 聚集方法，提供聚集大小
     */
    public int size()
    {
        return list.size();
    }
    /**
     * 辅助方法，向 DNS 服务器查询 MX 记录
     */
    private Vector getMXRecords(
        String providerUrl,
        String domainName) throws Exception
    {
        // 设置环境性质
        Hashtable env = new Hashtable();

        env.put(FACTORY_ENTRY, FACTORY_CLASS);
        env.put(PROVIDER_ENTRY, providerUrl);

        // 创建环境对象
        DirContext dirContext = new InitialDirContext(env);

        Vector records = new Vector(10);

        // 读取环境对象的属性
        Attributes attrs = dirContext.getAttributes(
            domainName,
            new String[] {MX_TYPE});

        for(NamingEnumeration ne = attrs.getAll();
            ne != null && ne.hasMoreElements();)
```

```
            {
                Attribute attr = (Attribute)ne.next();
                NamingEnumeration e = attr.getAll();
                while(e.hasMoreElements())
                {
                    String element = e.nextElement().toString();
                    StringTokenizer tokenizer =
                        new StringTokenizer(element, " ");

                    MailServer mailServer = new MailServer();

                    String token1 = tokenizer.nextToken();
                    String token2 = tokenizer.nextToken();

                    if(token1 != null && token2 != null)
                    {
                        mailServer.setPriority(
                            Integer.valueOf(token1).intValue());
                        mailServer.setServer(token2);
                        records.addElement(mailServer);
                    }
                }
            }
            System.out.println("List created.");
            return records;
        }
    }
```

在上面 getMXRecords()方法的源代码中使用了 Tokenizer，将 MX 记录按照空格将优先权和服务器的名字分开。

下面是一个示意性的客户端的源代码，如代码清单 9 所示。读者可以看到，这个客户端调用单例类 MXList 的静态工厂方法，将网络的 DNS 服务器名"dns01390.ny.jeffcorp.com"和所在的网域名"jeffcorp.com"以参量形式传入，便可以得到一个 MXList 类的实例。然后，客户端就可以使用这个实例的聚集方法一个个地读出所有的 SMTP 服务器信息。

代码清单 9：客户类的源代码

```
package com.javapatterns.singleton.mxrecord;
public class Client
{
    private static MXList mxl;
    public static void main(String[] args)
        throws Exception
    {
        mxl = MXList.getInstance(
            "dns://dns01390.ny.jeffcorp.com",
            "jeffcorp.com");
```

```
            for(int i=0; i < mxl.size(); i++)
            {
                System.out.println(
                    (1 + i)
                    + ") priority = "
                    + ((MailServer)
                    mxl.elementAt(i)).getPriority()
                    + ", Name = "
                    + ((MailServer)
                    mxl.elementAt(i)).getServer()
                    );
            }
        }
    }
```

由于这仅是一个示意性的客户端，因此，仅仅将查询的结果打印出来而已。在实际的应用中，读者可以将结果传给发送电子邮件的对象，从而控制向特定的 SMTP 服务器发送邮件的目的。

运行结果

程序在运行时会打印出一列 SMTP 服务器的名字以及它们的优先权。下面就是系统在本书作者的网络环境下运行的结果，如代码清单 10 所示。

代码清单 10：运行结果

```
1) priority = 10, Name = bird2-a.jeffcorp.com
2) priority = 10, Name = bird2-b.jeffcorp.com
3) priority = 10, Name = horse1-a.jeffcorp.com
4) priority = 10, Name = horse1-b.jeffcorp.com
5) priority = 10, Name = horse2-a.jeffcorp.com
6) priority = 10, Name = horse2-b.jeffcorp.com
7) priority = 10, Name = bird1-a.jeffcorp.com
8) priority = 10, Name = bird1-b.jeffcorp.com
```

16.6 讨　　论

发送邮件时的注意事项

在使用 JavaMail 包所提供的功能发送邮件时，要循环使用所有 MXList 给出的 SMTP 服务器进行邮件发送，同时注意捕捉被抛出的 javax.mail.SendFailedException 异常。如果没有捕捉到这个异常，就表明发送工作是成功的，应当终止循环；如果捕捉到这个异常，就表明发送工作失败，应当继续循环，尝试下一个 SMTP 服务器。

与多例模式的关系

最后值得指出的是，MXList 实际上是一个将单例模式应用到一个聚集上的例子。换言之，MXList 本身是一个聚集，向外界提供聚集管理方法，如 elementAt()和 size()等，而此聚集本身是一个单例。

熟悉多例模式的读者可能会问，这是不是与多例模式有相似之处？多例模式难道不是有一个聚集吗？

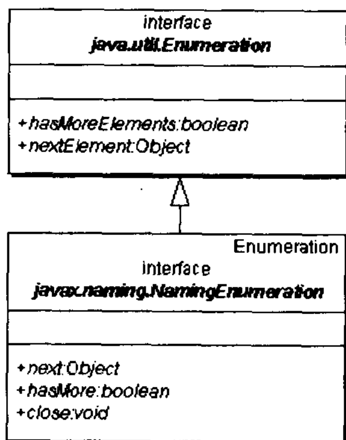
在多例模式中，聚集用来登记和管理多例类自身的实例，MXList 作为一个聚集，包含的元素是 MailServer 对象，而不是它自己。它们之间的关系就是一个多例对象与一个单例聚集对象的关系，两者虽然相像，但本质不同。

关于多例模式的详细讨论请参见本书的“专题：多例（Multiton）模式与多语言支持”一章。

迭代子模式的应用

关于迭代子模式的一般性讨论，请参见本书的“迭代子（Iterator）模式”一章。

Attributes 是一个聚集对象，因此它像所有的聚集一样，通过一个工厂方法提供一个迭代子对象。在这里工厂方法是 getAll()方法，而这个迭代子就是实现了 NamingEnumeration 接口的对象。NamingEnumeration 的类图如下图所示。



从上图可以看出，NamingEnumeration 接口是 Enumeration 接口的子接口，因为 NamingEnumeration 规定出一个新的 close()方法。

由于针对命名 - 目录服务的信息查询会占用大量的资源，因此，在一个 Naming - Enumeration 类型的对象被创建之后，一旦不再需要，就应当调用 close()方法将资源释放掉。如果一个 NamingEnumeration 类型的迭代子对象迭代到了最后的元素之后，也就是说当 hasMore()方法返还 false 时，就不必再调用 close()方法释放资源，因为迭代子对象会自动将占用的资源释放掉。反过来，如果迭代在没有到达最后一个元素之前就中止了，那么客户端应当明显地调用 close()方法以便释放资源。

需要注意的是，在调用了 close()方法后，这个迭代子对象就不能再使用了。如果客户端再次需要 NamingEnumeration 类型的对象，就必须调用工厂方法 getAll()，以便重新产生这个对象。

问答题

1. DNS 服务器的 MX 记录并不是真正静态的，能否修改一下上面的设计，让 XMList 类能够每隔 24 小时进行一次查询，而不是仅在初始化时做一次查询？

2. 可以使用 JNDI 查询 javax.sql.DataSource 吗？请给出源代码的关键部分。

3. 在上面的题目中使用了什么模式？

4. 请给出一个 EmailManager 类的源代码。这个类允许客户端传入收件人地址、发件人地址、邮件主题、邮件内容、附件等，并通过一个 SMTP 服务器发送电子邮件。

问答题答案

1. 这是可以的，只需要加入一个新的状态用于记录每一次 DNS 查询的时间即可。在用户调用 MX 记录表的时候，将实时与记录中的时间加以比较，如果时间差大于某一个数值时，系统就重新对 DNS 查询并重建 MX 记录表，然后再回应客户端的请求。

修改后的 XMList 类的源代码如代码清单 11 所示。

代码清单 11：XMList 类的源代码

```
package com.javapatterns.singleton.mxrecord1;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Date;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;
public class MXList
{
    private static MXList mxl = null;
    private Vector list = null;
    private static final String FACTORY_ENTRY =
        "java.naming.factory.initial";
    private static final String FACTORY_NAME =
        "com.sun.jndi.dns.DnsContextFactory";
    private static final String PROVIDER_ENTRY =
```

```
        "java.naming.provider.url";
    private static final String MX_TYPE = "MX";
    private String dnsUrl = null;
    private String domainName = null;
    private static Calendar updateTime ;
   /**
    * 默认构造子是私有的，
    * 保证本类不能被外部直接实例化
    */
    private MXList() { }
   /**
    * 构造子是私有的，
    * 保证本类不能被外部直接实例化
    */
    private MXList(String providerUrl,
        String domainName) throws Exception
    {
        this.dnsUrl = dnsUrl;
        this.domainName = domainName;

        this.list = getMXRecords(providerUrl,
            domainName);
    }
   /**
    * 静态工厂方法，提供本类惟一的实例
    */
    public static synchronized MXList getInstance(
        String providerUrl,
        String domainName) throws Exception
    {
        if (mxl == null)
        {
            mxl = new MXList(providerUrl, domainName);
        }
        else
        {
            // 计算现在时间与 MX 记录生成时间的间隔
            Calendar now = new GregorianCalendar();
            long TimeDifference = now.getTime().getTime()
                - updateTime.getTime().getTime() ;
            System.out.println("TimeDifference"
                + TimeDifference);

            // 如果距离 MX 记录生成的时间超过
            // 24 小时，则重新生成 MX 记录表
            if (TimeDifference  > 1000 * 60 * 60 * 24)
```

```
            {
                mxl.list = mxl.getMXRecords(
                    providerUrl, domainName);
            }
        }
        return mxl;
    }
    /**
     * 聚集方法，提供聚集元素
     */
    public MailServer elementAt(int index)
        throws Exception
    {
        return (MailServer) list.elementAt(index);
    }
    /**
     * 聚集方法，提供聚集大小
     */
    public int size()
    {
        return list.size();
    }
    /**
     * 辅助方法，向 DNS 服务器查询 MX 记录
     */
    private Vector getMXRecords(
        String dnsUrl,
        String domainName) throws Exception
    {
        // 设置环境性质
        Hashtable env = new Hashtable();

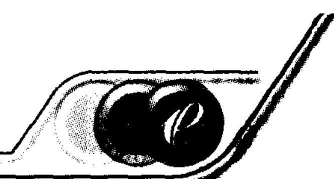
        env.put(FACTORY_ENTRY, FACTORY_NAME);
        env.put(PROVIDER_ENTRY, dnsUrl);

        // 创建环境对象
        DirContext dirContext = new InitialDirContext(env);

        Vector records = new Vector(10);

        // 取得环境对象的属性
        Attributes attrs = dirContext.getAttributes(
            domainName,
            new String[] {MX_TYPE});

        for(NamingEnumeration ae = attrs.getAll();
```



```
            ae != null && ae.hasMoreElements();)
        {
            Attribute attr = (Attribute)ae.next();
            NamingEnumeration e = attr.getAll();
            while(e.hasMoreElements())
            {
                String element = e.nextElement().toString();
                StringTokenizer tokenizer =
                    new StringTokenizer(element, " ");

                MailServer mailServer = new MailServer();

                String token1 = tokenizer.nextToken();
                String token2 = tokenizer.nextToken();

                if(token1 != null && token2 != null)
                {
                    mailServer.setPriority(
                        Integer.valueOf(token1).intValue());
                    mailServer.setServer(token2);
                    records.addElement(mailServer);
                }
            }
        }
        // 存储实时
        updateTime = new GregorianCalendar();
        System.out.println("List created.");
        return records;
    }
}
```

2. 这是可以的，代码清单 12 所示给出了查询 DataSource 的办法。

代码清单 12：查询 DataSource 对象的源代码

```
javax.naming.InitialContext ctx =
    new InitialContext();
Object resource = ctx.lookup("datasource");
javax.sql.DataSource hal =
    (javax.sql.DataSource) resource;
```

一旦 JNDI 返还了类型为 DataSource 的对象，就可以将这个对象当做正常的 DataSource 对象使用了。

3. 在上面的查询 DataSource 的代码中使用了工厂方法模式。lookup()方法就是工厂方法，resource 对象就是产品，而初始环境对象就是工厂角色。

相对于 java.sql.Connection 对象来说，DataSource 对象又是工厂角色，Connection 对象则是产品角色。

关于工厂方法模式，请参见本书的“工厂方法（Factory Method）模式”一章。

4. 由于篇幅限制，EmailManager 类的源代码不在这里给出。有兴趣的读者可以从本书的源代码库中看到这个类的源代码。

参考文献

[JNDI00] Sun Microsystem. Getting Started: The JNDI Overview. http://java.sun.com /products /jndi/tutorial/getStarted/overview/index.html

第 17 章　专题：多例（Multiton）模式与多语言支持

作为对象的创建模式，多例模式中的多例类可以有多个实例，而且多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

请读者在阅读本章之前，先阅读本书的“单例模式”一章。

17.1　引　　言

一个真实的项目

这是一个真实的面向全球消费者的华尔街金融网站项目的一部分。按照项目计划书，这个网站系统是要由数据库驱动的，并且要支持 19 种不同的语言，而且在将来支持更多的语言。消费者在登录到系统上时可以选择自己所需要的语言，系统则根据用户的选择将网站的静态文字和动态文字全部转换为用户所选择的语言。

经过讨论，设计师们同意对静态文字和动态文字采取不同的解决方案：

- 把所有的网页交给翻译公司对上面的静态文字进行翻译。
- 网页上面的动态内容需要程序解决。

在进行了研究后，设计师们发现，他们需要解决的动态文字的“翻译”问题，实际上是将数据库中的一些静态或者半静态的数据进行“翻译”。这里形成一个典型的数据表，如下表所示。

货币代码	货币名称	货币尾数
USD	America (United States of America), Dollars	2
CNY	China, Yuan Renminbi	2
EUR	France, Euro	2
JPY	Japan, Yen	0

货币代码永远是上面所看到的英文代码，但是货币名称应当根据用户所选择的语言不同而不同。比如对中文读者就应当翻译成为如下表所示的样子。

货币代码	货币名称	货币尾数
USD	美国（美利坚合众国），美元	2
CNY	中国，人民币元	2

（续表）

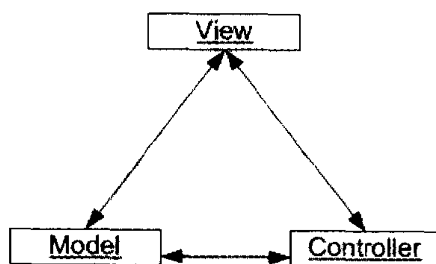
货币代码	货币名称	货币尾数
EUR	法国，欧元	2
JPY	日本，日元	0

这样的表会在网页上作为下拉菜单出现，用户看到的是货币名称，而系统内部使用的是货币代码。

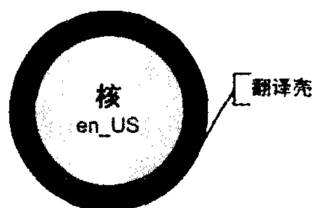
国际化解决方案

这样的问题就是国际化的问题，所谓国际化就是 Internationalization，简称 i18n（请参见本章最后的问答题）。

设计师所采取的实际方案是分层方案，也就是 MVC 模式。MVC 模式将系统分为三个层次，即模型（Model）、视图（View）、控制器（Control）。国际化是视图部分的问题，因此，应当在视图部分得到解决。MVC 模式的示意图如下图所示。



换言之，系统的内核可以是纯英文的。在内核外部增加一个壳层负责语言翻译工作。如右图所示，是英文内核和翻译壳层的概念图。



所谓内核就是系统的模型，而翻译壳层便是视图的一部分。对多语言的支持属于视图功能，因此，不应当在内核解决，而应当在视图解决，这就是设计师们达成的总体方案。

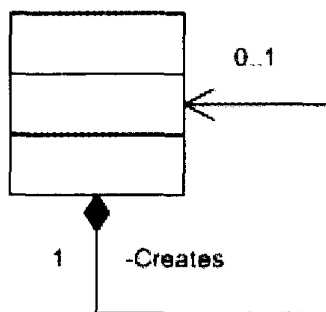
17.2 多例模式

多例模式的特点

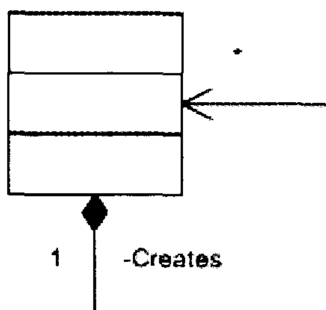
所谓的多例模式（Multiton Pattern），实际上就是单例模式的自然推广。作为对象的创建模式，多例模式或多例类有以下的特点：

- 多例类可有多个实例。
- 多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

单例类一般情况下最多只可以有一个实例，请见下图所示的单例类结构图。



但是单例模式的精神是允许有限个实例，并不是仅允许一个实例。这种允许有限多个实例并向整个 JVM 提供自己实例的类叫做多例类（Multiton），这种模式叫做多例模式（Multiton Pattern），请参见下图所示的多例类结构图。

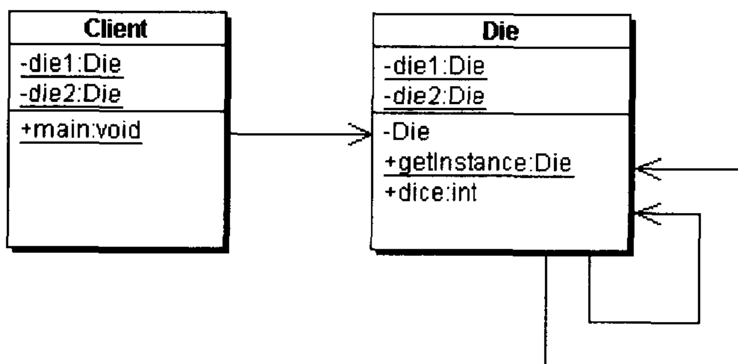


本章就需要用多例模式来实现资源对象，需要构造出能提供有限个实例、每个实例有各不相同的属性（即 Locale 代码）的代码。

有上限多例类

一个实例数目有上限的多例类已经把实例的上限当做逻辑的一部分，并建造到了多例类的内部，这种多例模式叫做有上限多例模式。

比如每一麻将牌局都需要两个骰子，因此骰子就应当是双态类。这里就以这个系统为例，说明多例模式的结构。骰子的类图如下图所示。



多例类 Die（骰子）的源代码如代码清单 1 所示。

代码清单 1：多例类的源代码

```
package com.javapatterns.multilingual.dice;
import java.util.Random;
import java.util.Date;
public class Die
{
    private static Die die1 = new Die();
    private static Die die2 = new Die();
   /**
    *     私有的构造子保证外界无法
    *     直接将此类实例化
    */
    private Die()
    {
    }
   /**
    *     工厂方法
    */
    public static Die getInstance(int whichOne)
    {
        if (whichOne == 1)
        {
            return die1;
        }
        else
        {
            return die2;
        }
    }
   /**
    *     掷骰子，返还一个在 1～6 之间的
    *     随机数
    */
    public synchronized int dice()
    {
        Date d = new Date();
        Random r = new Random( d.getTime() );
        int value = r.nextInt();
        value = Math.abs(value);
        value = value % 6;
        value += 1;
        return value;
    }
}
```

在多例类 Die 中，使用了饿汉方式创建了两个 Die 的实例。根据静态工厂方法的参数，工厂方法返还两个实例中的一个。Die 对象的 die()方法代表掷骰子，这个方法会返还一个在 1～6 之间的随机数，相当于骰子的点数，如代码清单 2 所示。

代码清单 2：客户端的源代码

```
package com.javapatterns.multilingual.dice;
public class Client
{
    private static Die die1, die2;
    public static void main(String[] args)
    {
        die1 = Die.getInstance(1);
        die2 = Die.getInstance(2);
        die1.dice();
        die2.dice();
    }
}
```

由于有上限的多例类对实例的数目有上限，因此有上限的多例类在这个上限等于 1 时，多例类就回到了单例类。因此，多例类是单例类的推广，而单例类是多例类的特殊情况。

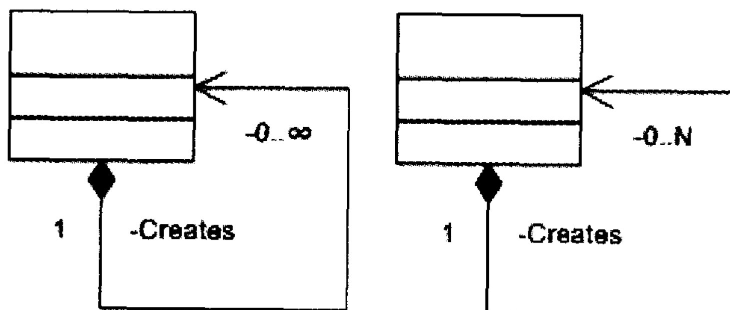
一个有上限的多例类可以使用静态变量存储所有的实例，特别是在实例数目不多的时候，可以使用一个个的静态变量存储一个个的实例。在数目较多的时候，就需要使用静态聚集存储这些实例。

无上限多例模式

多例类的实例数目并不需要有上限[CAMP02]，实例数目没有上限的多例模式就叫做无上限多例模式。

由于没有上限的多例类对实例的数目是没有限制的，因此，虽然这种多例模式是单例模式的推广，但是这种多例类并不一定能够回到单例类。

由于事先不知道要创建多少个实例，因此，必然使用聚集管理所有的实例。本章要讨论的多语言支持方案就需要应用到多例模式，关于没有上限的多例模式的实现可以参见下面的讨论。没有上限的多例模式（左）和有上限的多例模式（右）的类图如下图所示。



其中 N 就是实例数目的上限。

有状态的和没有状态的多例类

如同单例类可以分成有状态的和没有状态的两种一样，多例类也可以分成有状态的和没有状态的两种。

多例对象的状态如果是可以在加载后改变的，那么这种多例对象叫做可变多例对象（Mutable Singleton）；如果多例对象的状态在加载后不可以改变，那么这种多例对象叫做不变多例对象（Immutable Singleton）。显然不变多例类的情形较为简单，而可变单例类的情形较为复杂。

如果一个系统是建立在诸如 EJB 和 RMI 等分散技术之上的，那么多例类有可能会出现数个实例，因此，在这种情况下除非提供有效的协调机制，不然最好不要使用有状态的和可变的单例类，以避免出现状态不自恰的情况。读者可以参考本书的“单例（Singleton）模式”一章中的相关讨论。

17.3 多语言项目的设计

由于熟悉了多例模式，系统的设计实际上变得就不再复杂。

语言代码

几个常见的语言代码如下表所示。

语言代码	说明
de	German
en	English
fr	French
ja	Japanese
jw	Javanese
ko	Korean
zh	Chinese

地区代码

几个常见的地区代码如下表所示。

地区代码	说明
CN	China
DE	Germany
FR	France

（续表）

地区代码	说　明
IN	India
US	United States

Locale 代码

一个 Locale 代码由语言代码和地区代码组合而成，如下表所示。

语言代码	地区代码	Locale 代码	说　明
en	US	en_US	美国英语
en	GB	en_GB	英国英语
fr	FR	fr_FR	法国法语
fr	CA	fr_CA	加拿大法语
de	DE	de_DE	德国德语
zh	CH	zh_CH	简体汉语

Resource 文件及其命名规范

一个 Resource 文件是一个简单的文本文件。一个 Resource 文件的名字是由一个短文件名和文件的扩展名 properties 组成的，而 Resource 文件的短文件名则是 Java 程序在调用此文件时使用的文件名。

一个 Resource 文件和一个普通的 properties 文件并无本质区别，但 Java 语言对两者的支持是有区别的。java.util.Properties 类不支持多语言，而 java.util.ResourceBundle 类则支持多语言。

当 Locale 代码是 en_US 时，Resource 文件的文件名应当是短文件名加上 Locale 代码，就是 en_US。当 Locale 代码是 zh_CH 时，Resource 文件的文件名应当是短文件名加上 Locale 代码，就是 zh_CH。

怎样使用 Locale 对象和 ResourceBundle 对象

那么怎样使用 Local 对象和 ResourceBundle 读取一个 Resource 文件呢？如代码清单 3 所示。

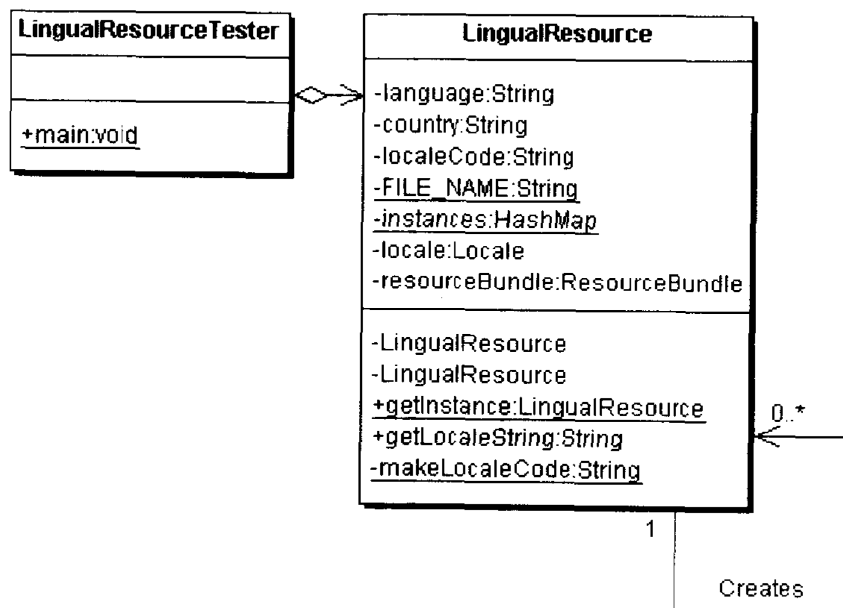
代码清单 3：怎样使用 Locale 对象和 ResourceBundle 对象

```
Locale locale = new Locale("fr","FR");
ResourceBundle res =
    ResourceBundle.getBundle("shortname",locale);
```

在上面的例子里面，res 对象会加载一个名为 shortname_fr_FR.properties 的 Resource 文件。

系统的设计

下图所示为系统的结构图。其中 LingualResourceTester 是一个示意性的客户端类，而 LingualResource 是一个多例类。



这个多例类的源代码如代码清单 4 所示。

代码清单 4：多例类 LingualResource 的源代码

```
package com.javapatterns.multilingual;
import java.util.HashMap;
import java.util.Locale;
import java.util.ResourceBundle;
public class LingualResource
{
    private String language = "en";
    private String region = "US";
    private String localeCode = "en_US";
    private static final String FILE_NAME = "res";
    private static HashMap instances =
        new HashMap(19);
    private Locale locale = null;
    private ResourceBundle resourceBundle = null;
    private LingualResource lnkLingualResource;
   /**
    *     私有的构造子保证外界无法直接将此类实例化
    */
    private LingualResource(
        String language, String region)
```



```
    {
        this.localeCode = language;
        this.region = region;
        localeCode =
            makeLocaleCode(language , region);
        locale = new Locale(language, region);
        resourceBundle =
            ResourceBundle.getBundle(FILE_NAME, locale);
        instances.put( makeLocaleCode(language, region) ,
            resourceBundle);
    }
   /**
    *   私有的构造子保证外界无法直接将此类实例化
    */
    private LingualResource()
    {
        //do nothing
    }
   /**
    *   工厂方法，返还一个具有指定的内部状态的实例
    */
    public synchronized static LingualResource
        getInstance(String language, String region)
    {
        if
            makeLocaleCode(language , region )))
        {
            return (LingualResource)
                makeLocaleCode(language , region ));
        }
        else
        {
            return new
                LingualResource(language, region);
        }
    }
    public String getLocaleString(String code)
    {
        return resourceBundle.getString(code);
    }
    private static String makeLocaleCode(
        String language, String region)
    {
        return language + "_" + region;
    }
}
```

其中的 makeLocaleCode()是一个辅助性的方法，在传入语言代码和地区代码时，此方法可以返回一个 Locale 代码。

这个多例类的构造子是私有的，因此不能用 new 关键字来实例化。所有的实例必须通过调用静态 getInstance()方法来得到。在 getInstance()方法被调用时，程序会首先检查传入的 Locale 代码是否已经在 instances 集合中存在，如果已经存在，即直接返回它所对应的 LingualResource 对象，否则就会首先创建一个此 Locale 代码所对应的 LingualResource 对象，将之存入 instances 集合，并返回这个实例。

下面给出一个客户端的源代码，如代码清单 5 所示。

代码清单 5：客户端类 LingualResourceTester 的源代码

```
package com.javapatterns.multilingual;
public class LingualResourceTester
{
    public static void main(String[] args)
    {
        LingualResource ling =
            LingualResource.getInstance("en" , "US");
        String usDollar = ling.getLocaleString("USD");
        System.out.println("USD=" + usDollar);
        LingualResource lingZh =
            LingualResource.getInstance("zh" , "CH");
        String usDollarZh = lingZh.getLocaleString("USD");
        System.out.println("USD=" + usDollarZh);
    }
}
```

如果用户是美国用户，那么在 JSP 网页中可以通过调用 getLocaleString()方法得到相应的英文说明，如代码清单 6 所示。

代码清单 6：Resource 文件 res_en_US.properties 的内容

```
LingualResource ling =
    LingualResource.getInstance("en" , "US");
String usDollar = ling.getLocaleString("USD");
```

就会返还：

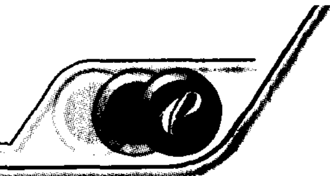
```
US Dollar
```

相应地，如果用户是中国大陆的用户，那么在 JSP 网页中可以通过调用 getLocaleString()方法得到相应的中文说明。比如：

```
LingualResource ling =
    LingualResource.getInstance("zh" , "CH");
String usDollar = ling.getLocaleString("USD");
```

就会返还：

```
美元
```



Resource 文件的内容

为美国英文准备的 Resource 文件 res_en_US.properties 的内容如下：

```
USD=US Dollar
JPY=Japanese Yen
```

为简体中文准备的 Resource 文件 res_zh_CH.properties 的内容如代码清单 7 所示。

代码清单 7：Resource 文件 res_zh_CH.properties 的内容

```
USD=美元
JPY=日元
```

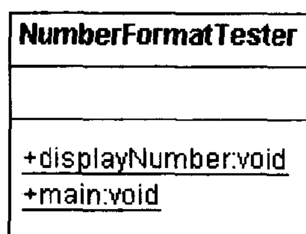
问答题

1. 请问为什么 Internationalization 又简称做 i18n？
2. 请给出一个根据语言代码和地区代码将数字格式化的例子。
3. 请给出一个根据语言代码和地区代码将货币数字格式化的例子。
4. 请给出一个根据语言代码和地区代码将百分比格式化的例子。

问答题答案

1. 在英文字 Internationalization 中，第一个字母 i 和最后一个字母 n 之间有 18 个字母，因此，Internationalization 又简称做 i18n。

2. Java 库 java.text.NumberFormat 类提供了对数字格式的支持，下面给出的就是对数字格式支持的解答的类图，如下图所示。



程序的源代码如代码清单 8 所示。

代码清单 8：Resource 文件 res_zh_CH.properties 的内容

```
package com.javapatterns.multilingual.number;
import java.util.Locale;
import java.text.NumberFormat;
public class NumberFormatTester
{
    static public void displayNumber(
        Double amount, Locale currentLocale)
```

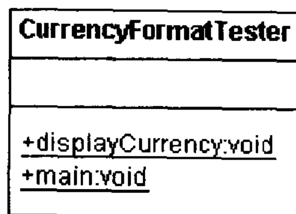
```
    {
        NumberFormat formatter;
        String amountOut;
        formatter =
          NumberFormat.getNumberInstance(currentLocale);
        amountOut = formatter.format(amount);
        System.out.println(amountOut + "     "
             + currentLocale.toString());
    }
    static public void main(String[] args)
    {
        displayNumber(new Double(1234567.89),
             new Locale("en", "US"));
        displayNumber(new Double(1234567.89),
             new Locale("de", "DE"));
        displayNumber(new Double(1234567.89),
             new Locale("fr", "FR"));
    }
}
```

在运行时，程序打印出的结果如代码清单 9 所示。

代码清单 9：Resource 文件 res_zh_CH.properties 的内容

```
456,789%    en_US
456.789%    de_DE
456 789%    fr_FR
```

3. Java 库 java.text.NumberFormat 类提供了对货币数字格式的支持。下面给出的就是对货币数字格式支持的解答的类图。



其源代码如代码清单 10 所示。

代码清单 10：Resource 文件 res_zh_CH.properties 的内容

```
package com.javapatterns.multilingual.number;
import java.util.Locale;
import java.text.NumberFormat;
public class CurrencyFormatTester
{
    static public void displayCurrency(Double amount,
         Locale currentLocale)
    {
```



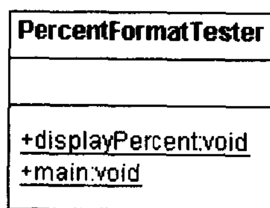
```
        NumberFormat formatter;
        String amountOut;
        formatter =
         NumberFormat.getCurrencyInstance(currentLocale);
        amountOut = formatter.format(amount);
        System.out.println(amountOut + "     "
            + currentLocale.toString());
    }
    static public void main(String[] args)
    {
        displayCurrency(new Double(1234567.89),
            new Locale("en", "US"));
        displayCurrency(new Double(1234567.89),
            new Locale("de", "DE"));
        displayCurrency(new Double(1234567.89),
            new Locale("fr", "FR"));
    }
}
```

在运行时，程序打印出的结果，如代码清单 11 所示。

代码清单 11：Resource 文件 res_zh_CH.properties 的内容

```
$1,234,567.89       en_US
1.234.567,89 DM     de_DE
1 234 567,89 F      fr_FR
```

4. Java 库 java.text.NumberFormat 类提供了对百分比格式的支持，下图所示就是对百分比格式支持的解答的类图。



其源代码如代码清单 12 所示。

代码清单 12：Resource 文件 res_zh_CH.properties 的内容

```
package com.javapatterns.multilingual.number;
import java.util.Locale;
import java.text.NumberFormat;
public class PercentFormatTester
{
    static public void displayPercent(
        Double amount, Locale currentLocale)
    {
        NumberFormat formatter;
```

```
        String amountOut;
        formatter =
         NumberFormat.getPercentInstance(currentLocale);
        amountOut = formatter.format(amount);
        System.out.println(amountOut + "     "
            + currentLocale.toString());
    }
    static public void main(String[] args)
    {
        displayPercent(new Double(4567.89),
            new Locale("en", "US"));
        displayPercent(new Double(4567.89),
            new Locale("de", "DE"));
        displayPercent(new Double(4567.89),
            new Locale("fr", "FR"));
    }
}
```

在运行时，程序打印出的结果如代码清单 13 所示。

代码清单 13：Resource 文件 res_zh_CH.properties 的内容

```
1,234,567.89    en_US
1.234.567,89    de_DE
1 234 567,89    fr_FR
```

（本章问答题的第 2、3、4 题的解答参考了[GREEN]的相关例子，本书做了一些改动）

参考文献

[GREEN] Dale Green. Internationalization. http://java.sun.com/docs/books/tutorial/trailmap.html

[ISO639] International Organization for Standardization. ISO 639 Language Codes. http://www.iso.org

[ISO3166] International Organization for Standardization. ISO 3166 Country Codes. http://www.iso.org

[ISO4217] International Organization for Standardization. ISO 4217 Currency Codes. http://www.iso.org

[CAMP02] David Van Camp. The Pattern Digest. http://patterndigest.tripod.com/

第 18 章　专题：序列键生成器与单例及多例模式

本章的内容来自于一个真实的全球金融网站项目，本书去掉了所有与商业有关的内容，仅仅讨论技术观点。

18.1　问　　题

序列键

开发过数据库驱动信息系统的读者都知道，在一个关系数据库中，所有的数据都是存储在表里的，而现代的数据库设计要求每一个表都有一个主键（Primary Key）。对大多数的用户输入数据来讲，主键需要由系统以序列号方式产生，而不是由操作人员给出。

有一些关系数据库引擎提供某种序列键生成机制。比如，Microsoft SQL Server 提供一个 AutoNumber 的属性，允许每一个表内可以有一个 AutoNumber 列。Oracle 提供 Sequence 对象，可以提供序列键值。下面的 SQL 语句会建立一个名为 ISSUE_SEQ 的 Sequence 对象，其开始值为 1 000，每次增加 1，如代码清单 1 所示。

代码清单 1：用来创建一个 Oracle 序列的 SQL 语句源代码

```
CREATE SEQUENCE ISSUE_SEQ
 INCREMENT BY 1
 START WITH 1000
 MINVALUE 20
 MAXVALUE 9999999999999999999999999999
 NOCYCLE
 NOORDER
NOCACHE
```

其他一些数据库引擎则没有相应的机制，比如 Sybase 就没有类似的功能。这时候就只好使用一个表，并在其内部设有两个列，一个列存放键名，另一个列存放键值，客户端使用 SQL 语句自行管理键值。使用一个表来存储所有的序列键，如下表所示。

KeyName	KeyValue
PO_NUMBER	1045
SO_NUMBER	5069

（续表）

KeyName	KeyValue
ITEM_NUMBER	10874
RMA_NUMBER	2065
…	

有些商业化的系统（如 Vignette Story Server）需要支持几种主要的数据库，包括 Oracle，Microsoft SQL Server 和 Sybase 等。这样的系统就不能使用 Microsoft SQL Server 或者 Oracle 特有的机制，而必须使用某种具有普遍性的机制。这时，上面提到的使用一个表来管理各种主键就变得较为合理。

预定式存储

不论是使用哪一个机制，最终系统必须要由一些数据库操作来提供这些序列值。比如一个餐馆的贩卖系统需要一个序列号给每天开出去的卖单编号，这个序列号码就应当存放到数据库里面。每当发出序列号码的时候，都应当从数据库读取这个号码，并更新这个号码。

为了保证在任何情况下键值都不会出现重复，应当使用预定式键值存储办法。在请求一个键值时，首先将数据库中的键值更新为下一个可用值，然后将旧值提供给客户端。这样万一出现运行中断的话，最多就是这个键值被浪费掉。

与此相对的是记录式键值存储办法。也就是说，键值首先被返还给客户端，然后记录到数据库中去。这样做的缺点是，一旦系统中断，就有可能出现客户端已经使用了一个键值，而这个键值却没有来得及存储到数据库中的情况。在系统重启之后，系统还会从这个已经被使用过的键值开始，从而导致错误。因此不要使用这种登记式的存储办法。

预定式的存储办法可以每一次预定多个键值（也即一个键值区间），而不是每一次仅仅预定一个值。由于这些值都是一些序列数值，因此，所谓一次预定多个值，不过就是每次更新键值时将键值增加一个大于 1 的数目。在后面的设计方案中，首先考虑每次预定一个键值的做法，然后将之改进为每次预定 20 个值的情况。

单例模式的应用

上面讨论了序列的存储机制，另一个重要的机制是键的查询管理机制。与其将键值的查询工作交给各个模块，不如将之集中到一个对象身上。这个对象负责管理序列键的查询，称之为序列键管理器。

显然，不难看出，整个系统只需要一个序列键管理器对象。由于系统运行期间总是需要序列键，因此序列键管理器对象需要在系统运行期间存在。考虑到可以让一个序列键管理器负责管理分属于不同模块的多个序列键，因此这个序列键管理器需要让整个系统访问。

学习过单例模式的读者会意识到，这个系统设计应当使用到单例模式。是的，这个序列键管理器可以设计成一个单例类。

一个客户端系统往往需要管理不止一个键值，而是多个键值。这时候，可以将这个单例对象的内部状态扩展成为一个聚集，从而可以存储任意多个键值。也就是说，这个序列键管理器是一个聚集对象，而此聚集本身是一个单例对象。

关于单例模式，请读者参考本书的“单例（Singleton）模式”一章。

多例模式的应用

多例模式往往持有一个内蕴状态，多例类的每一个实例都有独特的内蕴状态。一个多例类持有一个聚集对象，用来登记自身的实例，而其内蕴状态往往就是登记的键值。当客户端通过多例类的静态工厂方法请求多例类的实例时，这个工厂方法都会在聚集内查询是否已经有一个这样的实例。如果有，就直接返还给客户端；如果没有，就首先创建一个这样的实例，将之登记到聚集中，然后再向客户端提供。

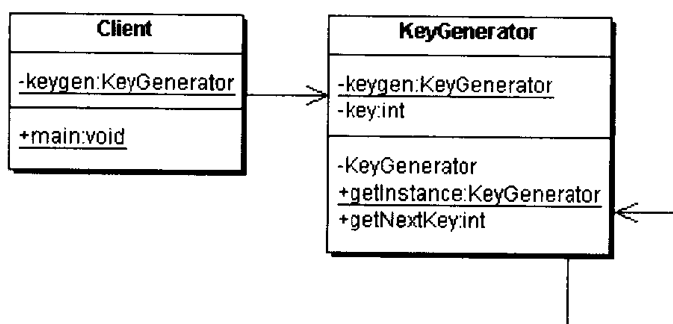
关于多例模式以及它与单例模式的关系，请读者参考本书的“专题：多例（Multiton）模式与多语言支持”一章。

18.2　将单例模式应用到系统设计中

下面从一个最简单的情况出发，逐渐将问题的复杂性提高，直到给出具有实用价值的解决方案为止。

方案一：没有数据库的情况

首先考虑一个没有数据库背景的方案，这个设计由一个单例类 KeyGenerator 组成。序列键生成器的类图如下图所示。



下面是这个键生成器 KeyGenerator 的源代码，如代码清单 2 所示。

代码清单 2：KeyGenerator 类的源代码

```
package com.javapatterns.keygen.ver1;
public class KeyGenerator
{
    private static KeyGenerator keygen =
        new KeyGenerator();
```

```
    private int key = 1000;
   /**
    *  私有构造子，保证外界无法直接实例化
    */
    private KeyGenerator() { }
   /**
    *  静态工厂方法，提供自己的实例
    */
    public static KeyGenerator getInstance()
    {
        return keygen;
    }
   /**
    *  取值方法，提供下一个合适的键值
    */
    public synchronized int getNextKey()
    {
        return key++;
    }
}
```

可以看出，上面的 KeyGenerator 类的构造子是私有的，因此，外界无法通过调用构造子将之实例化。同时，它提供了一个静态的工厂方法 getInstance()，自己向外界提供自己的实例。如果再考察一下这个工厂方法就会发现，这个方法永远仅提供同一个实例。换言之，这是一个单例类。

商业方法 getNextKey()返还一个整型数，这个数会自行递增，每次加 1，如代码清单 3 所示。

代码清单 3：客户类的源代码

```
package com.javapatterns.keygen.ver1;
public class Client
{
    private static KeyGenerator keygen;
    public static void main(String[] args)
    {
        keygen = KeyGenerator.getInstance();
        System.out.println("key = " + keygen.getNextKey());
        System.out.println("key = " + keygen.getNextKey());
        System.out.println("key = " + keygen.getNextKey());
    }
}
```

在运行时，客户对象会打印出得到的序列键的数值，这表明系统是正常工作的，如代码清单 4 所示。

代码清单 4：运行结果

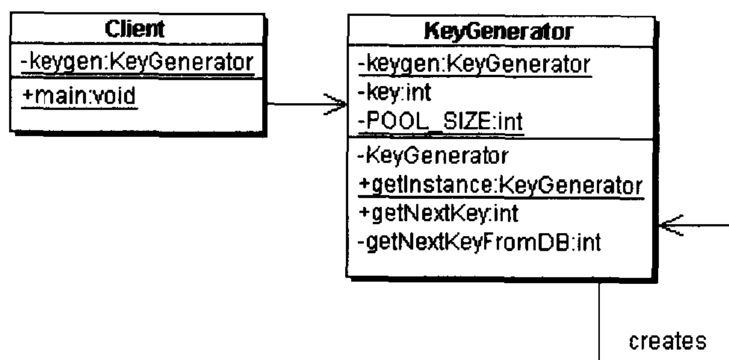
```
key = 1000
key = 1001
key = 1002
```

这一设计基本上实现了向客户端提供键值的功能，但是也有明显的缺点。由于没有数据库的存储，一旦系统重新启动，KeyGenerator 都会重新初始化，这就会造成键值的重复。

为了避免这一点，就必须将每次的键值存储起来，以便一旦系统中断和重启时，可以将这个键值取出，并在这个值的基础上重新开始。这就将设计师引向了下一个设计方案。

方案二：有数据库的情况

这个方案是对方案一的修正。与方案一一样，这个设计由一个单例类组成；而与方案一不同的是，这个单例类有数据库功能。它将键值存储在数据库的表中，每次客户端请求键值时，首先将这个表中的值增加 1，然后将这个值返还给客户端（当然，这两个数据库操作应当是一个完整的交易单位）。有数据库的键值生成器的结构图如下图所示。



下面就是 KeyGenerator 类的源代码，如代码清单 5 所示。这是一个单例类，它提供了私有的构造子，所以外界无法直接将其实例化，所有的实例化请求都必须通过静态工厂方法进行。

代码清单 5：KeyGenerator 的源代码

```
package com.javapatterns.keygen.ver2;
public class KeyGenerator
{
    private static KeyGenerator keygen =
        new KeyGenerator();
   /**
    *  私有构造子，保证外界无法直接实例化
    */
    private KeyGenerator() { }
   /**
    *  静态工厂方法，提供自己的实例
```

```
        */
        public static KeyGenerator getInstance()
        {
            return keygen;
        }
       /**
        *  取值方法，提供下一个合适的键值
        */
        public synchronized int getNextKey()
        {
            return getNextKeyFromDB();
        }
        private int getNextKeyFromDB()
        {
            String sql1 = "UPDATE KeyTable SET keyValue = keyValue + 1 ";
            String sql2 = "SELECT keyValue FROM KeyTable";
            //execute the update SQL
            //run the SELECT query
            //示意性地返还一个数值
            return 1000;
        }
    }
```

在接到客户端的请求时，这个 KeyGenerator 每次都向数据库查询键值，将新的键值登记到表里，然后将查询的结果返还给客户端。在上面的源代码中，给出了两个 SQL 语句，但是并没有给出执行这两行语句的 JDBC 代码。相信读者在将这个设计应用到自己的系统中时，可以将必要的 JDBC 代码加进去。

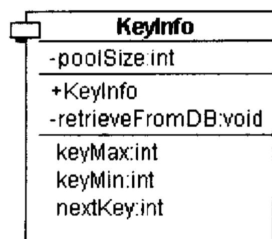
必须指出的是，为了将读者的注意力集中在系统设计上面，本书不想涉及到 JDBC 的细节，因此上面并没有给出这方面的源代码。相信读者在将这个设计应用到自己的系统中去时，可以自行实现这部分代码。

在这个设计方案里面，可以使用与第一个设计方案相同的客户端，因此就不在此重复了。

方案三：键值的缓存方案

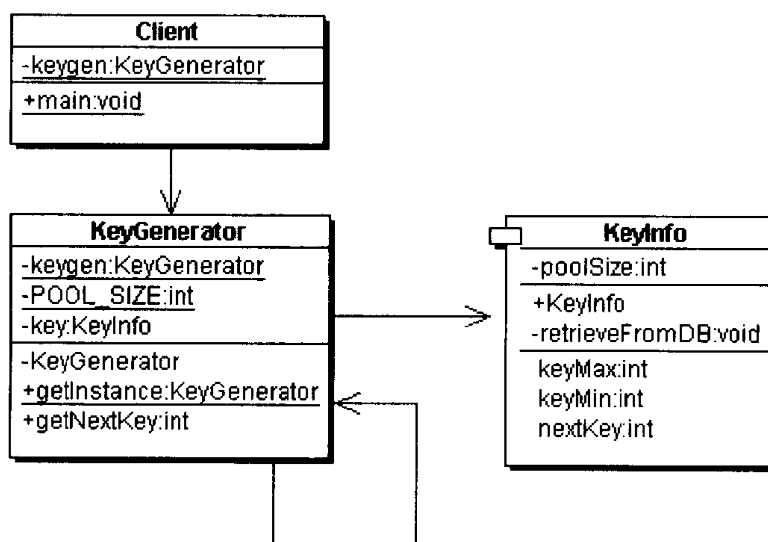
每一次都进行键值的查询，有必要吗？毕竟一个键的值只是一些序列号码，与其每接到一次请求就查询一次，然后向客户端提供这一个值，不如在一次查询中一次性地预先登记多个键值，然后连续多次地向客户端提供这些预订的键值。这样一来，不是节省了大部分不必要的数据库查询操作吗？

是的，这就是键值的缓存机制。当 KeyGenerator 每次更新数据库中的键值时，它都将键值增加。与方案二不同之处是，键值的增加值不是 1 而是更多。在下面给出的例子中，键值的增加值是 20。为了存储所有的与键有关的信息，特地引进一个 KeyInfo 类，如下图所示。



这个 KeyInfo 除了存储与键有关的信息外，还提供了一个 retrieveFromDB()方法，向数据库查询键值。每次查询得到的 20 个键值会在随后提供给请求者，直到 20 个键值全部使用完毕，然后再向数据库预定后 20 个键值。

KeyGenerator 作为一个单例类，保持一个对 KeyInfo 对象的引用。客户端调用 getNextKey()方法以得到下一个键的键值。有缓存的序列键生成机制如下图所示。



下面就是 KeyGenerator 类的源代码，如代码清单 6 所示。与第二个方案一样，KeyGenerator 类使用了私有的构造子，以及一个静态工厂方法向外界提供自己惟一的实例。

代码清单 6：KeyGenerator 的源代码

```
package com.javapatterns.keygen.ver3;
public class KeyGenerator
{
    private static KeyGenerator keygen =
        new KeyGenerator();
    private static final int POOL_SIZE = 20;
    private KeyInfo key ;
   /**
    *  私有构造子，保证外界无法直接实例化
    */
    private KeyGenerator()
    {
```

```
        key = new KeyInfo(POOL_SIZE);
    }
    /**
     *  静态工厂方法，提供自己的实例
     */
    public static KeyGenerator getInstance()
    {
        return keygen;
    }
    /**
     *  取值方法，提供下一个合适的键值
     */
    public synchronized int getNextKey()
    {
        return key.getNextKey();
    }
}
```

下面是 KeyInfo 类的源代码，如代码清单 7 所示。正如同上面所谈到的，这个类提供了向数据库查询的功能，并且存储一定数目的键值。

代码清单 7：KeyInfo 的源代码

```
package com.javapatterns.keygen.ver3;
class KeyInfo
{
    private int keyMax;
    private int keyMin;
    private int nextKey;
    private int poolSize;
    /**
     *  构造子
     */
    public KeyInfo(int poolSize)
    {
        this.poolSize = poolSize;
        retrieveFromDB();
    }
    /**
     *  取值方法，提供键的最大值
     */
    public int getKeyMax()
    {
        return keyMax;
    }
    /**
     *  取值方法，提供键的最小值
     */
```

```
    public int getKeyMin()
    {
        return keyMin;
    }
    /**
     *  取值方法，提供键的当前值
     */
    public int getNextKey()
    {
        if (nextKey > keyMax)
        {
            retrieveFromDB();
        }
        return nextKey++;
    }
    /**
     *  内部方法，从数据库提取键的当前值
     */
    private void retrieveFromDB()
    {
        String sql1 = "UPDATE KeyTable SET keyValue = keyValue + "
            + poolSize + " WHERE keyName = 'PO_NUMBER'";
        String sql2 = "SELECT keyValue FROM KeyTable WHERE KeyName = 'PO_NUMBER'";
        // execute the above queries in a transaction and commit it
        // assume the value returned is 1000
        //示意性地返还一个数值
        int keyFromDB = 1000;
        keyMax = keyFromDB;
        keyMin = keyFromDB - poolSize + 1;
        nextKey = keyMin;
    }
}
```

下面就是一个示意性的客户端的源代码，如代码清单 8 所示。

代码清单 8：一个示意性的客户端的源代码

```
package com.javapatterns.keygen.ver3;
public class Client
{
    private static KeyGenerator keygen;
    public static void main(String[] args)
    {
        keygen = KeyGenerator.getInstance();
        for (int i = 0 ; i < 20 ; i++)
        {
            System.out.println("key(" + (i+1)
                + ")= " + keygen.getNextKey());
```

```
        }
    }
}
```

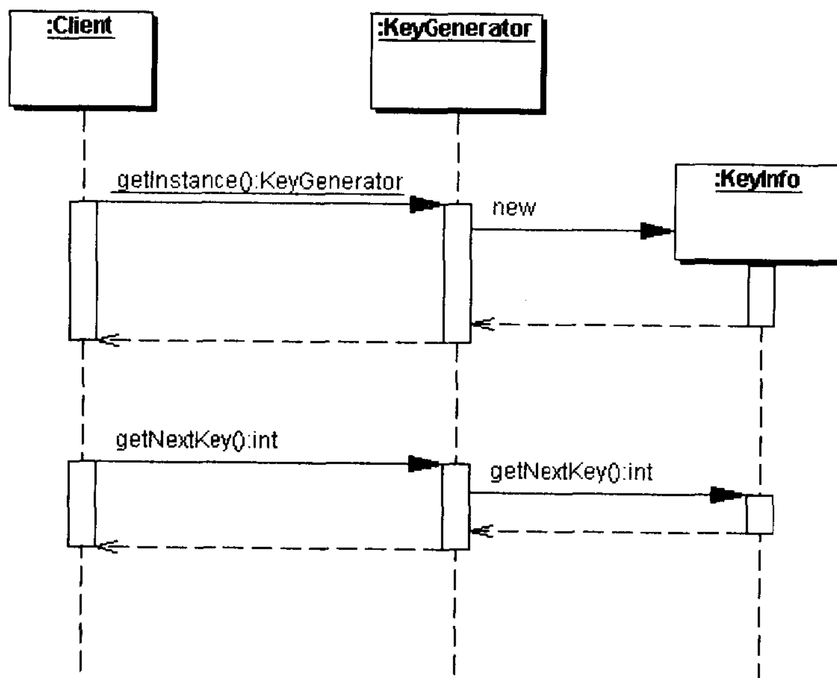
可以看出，这个示意性的客户端首先通过调用 KeyGenerator 的静态工厂方法得到 KeyGenerator 的实例，然后调用 KeyGenerator 对象的取值方法，以得到一个个的键值。运行的结果如代码清单 9 所示。

代码清单 9：系统的运行结果

```
key(1)= 981
key(2)= 982
key(3)= 983
......
key(19)= 999
key(20)= 1000
```

为了说明系统的活动时序，这里特地给出系统的活动序列图，如下图所示。从图中可以看出，客户端首先通过调用 KeyGenerator 的静态工厂方法 getInstance()得到 KeyGenerator 的单例实例，与此同时 KeyInfo 对象被创建。

然后，客户端调用 KeyGenerator 对象的 getNextKey()方法，而 KeyGenerator 对象则将调用委派给 KeyInfo 对象的 getNextKey()方法。KeyInfo 对象则通过数据库调用，将查询所得的键值返还给客户端。



现在，这个键值生成器已经具有如下的功能：在整个系统中是惟一的，能将生成的键值存储到数据库中，以便在系统重新启动时也能够继续键值的生成，而不会造成键值上的重复。

这本来已经足够好了，但是还有一点值得设计师考虑改进的是，一般的系统都不会只

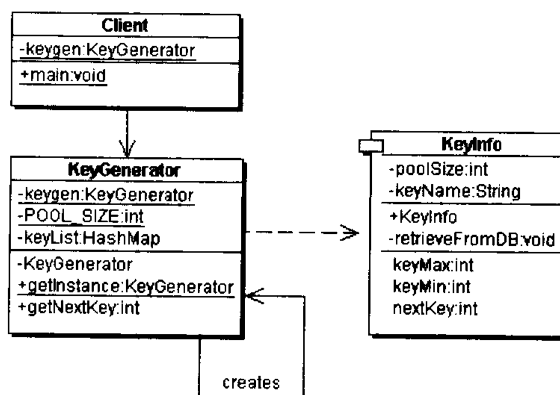
有一个键值，而是有多个键值需要生成。怎么让上面的设计适用于任意多个键值的情况呢？

首先，由于 KeyGenerator 是单例类，因此，给出多个 KeyGenerator 的实例并无可能，除非将之推广为多例类。

其次，虽然 KeyGenerator 是单例类，但是 KeyGenerator 仍然可以在内部使用一个聚集管理多个键值。换言之，可以使用一个本身是单例对象的聚集对象，配合上合适的接口达到目的。下面本书就首先考虑这一方案。

方案四：有缓存的多序列键生成器

方案四是对方案三的改进。在本方案中，同样使用 KeyInfo 对象存储某一个键的信息。与方案三相比，本方案引进了一个聚集用来存储不同序列键信息的 KeyInfo 对象，如下图所示。



可以看出，KeyGenerator 类仍然是一个单例类。它提供了私有的构造子和一个静态工厂方法，向外界提供自己惟一的实例。

下面就是 KeyGenerator 类的源代码，如代码清单 10 所示。

代码清单 10：KeyGenerator 类的源代码

```
package com.javapatterns.keygen.ver4;
import java.util.HashMap;
public class KeyGenerator
{
    private static KeyGenerator keygen =
        new KeyGenerator();
    private static final int POOL_SIZE = 20;
    private HashMap keyList = new HashMap(10);
  /**
   *  私有构造子，保证外界无法直接实例化
   */
   private KeyGenerator()
   {
   }
  /**
```

```
    *  静态工厂方法，提供自己的实例
    */
    public static KeyGenerator getInstance()
    {
        return keygen;
    }
   /**
    *  取值方法，提供下一个合适的键值
    */
    public synchronized int getNextKey(String keyName)
    {
        KeyInfo keyinfo ;
        if ( keyList.containsKey(keyName) )
        {
            keyinfo = (KeyInfo) keyList.get(keyName);
            System.out.println("key found");
        }
        else
        {
            keyinfo = new KeyInfo(POOL_SIZE, keyName);
            keyList.put(keyName, keyinfo);
            System.out.println("new key created");
        }
        return keyinfo.getNextKey(keyName);
    }
}
```

下面就是 KeyInfo 类的源代码，如代码清单 11 所示。这个类存储了键名、缓冲的大小及在这个缓冲区内的最小键值、最大键值、当前键值等信息。这个类还提供了一个 retrieveFromDB()方法，向数据库查询键值。

代码清单 11：KeyInfo 的源代码

```
package com.javapatterns.keygen.ver4;
class KeyInfo
{
    private int keyMax;
    private int keyMin;
    private int nextKey;
    private int poolSize;
    private String keyName;
   /**
    *  构造子
    */
    public KeyInfo(int poolSize, String keyName)
    {
        this.poolSize = poolSize;
```

```
        this.keyName = keyName;
        retrieveFromDB();
    }
    /**
     * 取值方法，提供键的最大值
     */
    public int getKeyMax()
    {
        return keyMax;
    }
    /**
     * 取值方法，提供键的最小值
     */
    public int getKeyMin()
    {
        return keyMin;
    }
    /**
     * 取值方法，提供键的当前值
     */
    public int getNextKey()
    {
        if (nextKey > keyMax)
        {
            retrieveFromDB();
        }
        return nextKey++;
    }
    /**
     * 内部方法，从数据库提取键的当前值
     */
    private void retrieveFromDB()
    {
        String sql1 = "UPDATE KeyTable SET keyValue = keyValue + "
            + poolSize + " WHERE keyName = '"
            + keyName + "'";
        String sql2 = "SELECT keyValue FROM KeyTable WHERE KeyName = '"
            + keyName + "'";
        // execute the above queries in a transaction and commit it
        // assume the value returned is 1000
        int keyFromDB = 1000;
        keyMax = keyFromDB;
        keyMin = keyFromDB - poolSize + 1;
        nextKey = keyMin;
    }
}
```

从上面的源代码可以看出，每当 getNextKey()被调用时，这个方法都会根据缓冲区的大小和已经用过的键值来判断是否需要更新缓冲区。当缓冲区被更新后，KeyInfo 会持有已经向数据库预定过的 20 个序列号码，并不断向调用者顺序提供这 20 个号码。等这 20 个序列号码用完之后，KeyInfo 对象就会向数据库预定后 20 个新号码。

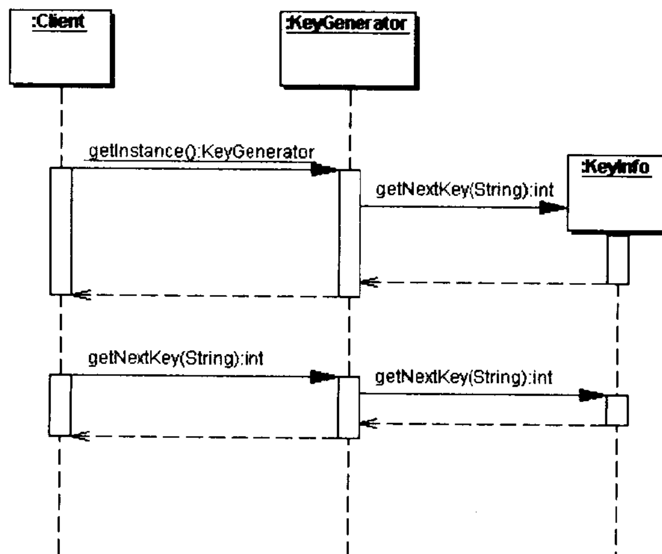
当然，如果系统被重新启动，而缓冲区中的号码并没有用完的话，这些没有用完的号码就不会再次被使用了。系统重新启动之后，KeyInfo 对象会重新向数据库预定下面的 20 个号码，并向外界提供这 20 个号码。

下面就是一个示意性的客户端 Client 类的源代码，如代码清单 12 所示。

代码清单 12：客户端的源代码

```
package com.javapatterns.keygen.ver4;
public class Client
{
    private static KeyGenerator keygen;
    public static void main(String[] args)
    {
        keygen = KeyGenerator.getInstance();
        for (int i = 0 ; i < 20 ; i++)
        {
            System.out.println("key(" + (i+1)
                + ")= " + keygen.getNextKey("PO_NUMBER"));
        }
    }
}
```

为了说明系统的活动时序，这里特地给出系统的活动序列图，如下图所示。从图中可以看出，客户端首先通过调用 KeyGenerator 的静态工厂方法 getInstance()得到 KeyGenerator 的单例实例，与此同时 KeyInfo 对象被创建。与方案三的情况不同的是，这里的 KeyInfo 的构造子接受键名作为参量。

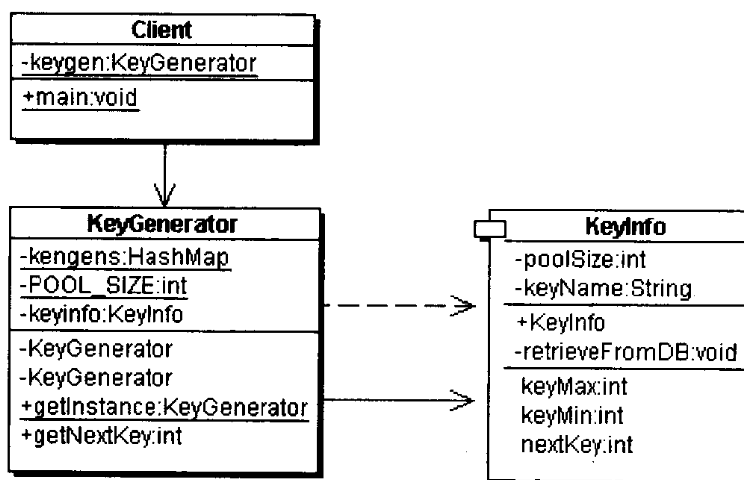


然后客户端调用 KeyGenerator 对象的 getNextKey()方法，而 KeyGenerator 对象则将调用委派给 KeyInfo 对象的 getNextKey()方法。KeyInfo 对象则通过数据库调用，将查询所得的键值返还给客户端。与方案三的情况不同的是，这里的两个 getNextKey()方法都接受键名作为参量。

运行的结果与上一个设计方案类似，所以不再重复。

18.3　将多例模式应用到系统设计中

正如前面所谈到的，为了能够处理多系列键值的情况，除了可以将单例模式所封装的单一状态改为聚集状态之外，还可以采用多例模式。多例模式允许一个类有多个实例，这些实例有各自不同的内蕴状态。这就是本书给出的第五个方案，应用多例模式的设计方案。下图所示就是这个设计方案的类图结构。



下面是 KeyGenerator 类的源代码，如代码清单 13 所示。可以看出，这是一个多例类，每一个 KeyGenerator 对象都持有一个特定的 KeyInfo 对象作为内蕴状态。客户端可以使用这个类的静态工厂方法得到所需要的实例，而这个工厂方法会首先查看做登记用的 keygens 聚集。如果所要求的键名在聚集里面，就直接将这个键名所对应的实例返还给客户端；如果所要求的键名不在聚集里面，就需要创建一个新的实例，对应于这个键名，然后将这个实例登记到聚集里面，再返还给客户端。

代码清单 13：KeyGenerator 类的源代码

```
package com.javapatterns.keygen.ver5;
import java.util.HashMap;
public class KeyGenerator
{
    private static HashMap kengens
        = new HashMap(10);
    private static final int POOL_SIZE = 20;
    private KeyInfo keyinfo;
```

```
    /**
     *  私有构造子，保证外界无法直接实例化
     */
    private KeyGenerator()
    {
    }
    /**
     *  私有构造子，保证外界无法直接实例化
     */
    private KeyGenerator(String keyName)
    {
        keyinfo = new KeyInfo(POOL_SIZE, keyName);
    }
    /**
     *   静态工厂方法，提供自己的实例
     */
    public static synchronized KeyGenerator
        getInstance(String keyName)
    {
        KeyGenerator keygen;
        if (kengens.containsKey(keyName))
        {
            keygen = (KeyGenerator)
                kengens.get(keyName);
        }
        else
        {
            keygen = new KeyGenerator(keyName);
        }
        return keygen;
    }
    /**
     *   取值方法，提供下一个合适的键值
     */
    public synchronized int getNextKey()
    {
        return keyinfo.getNextKey();
    }
}
```

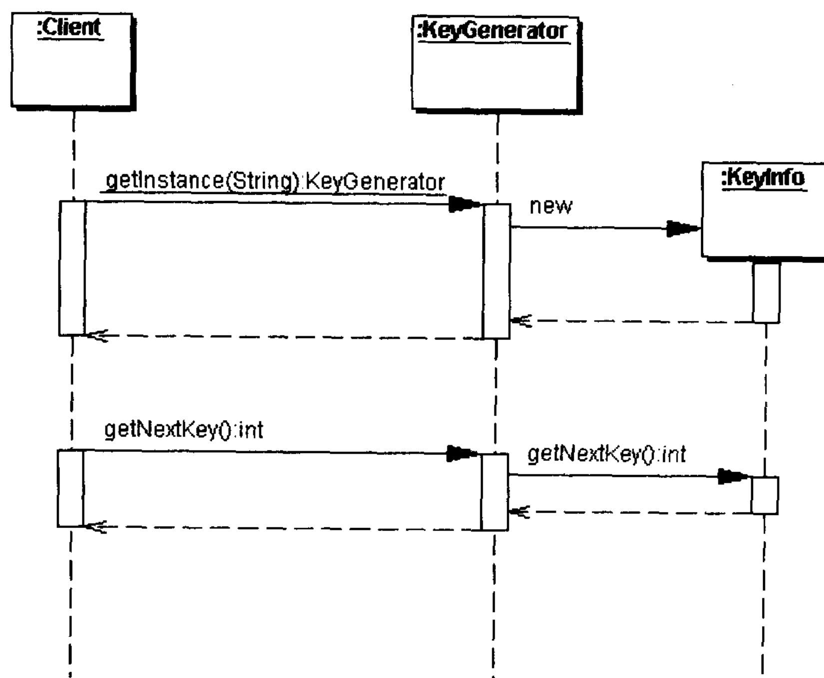
在这个设计里面 KeyInfo 类与设计方案四中的 KeyInfo 类并无区别，因而省略。

下面是一个示意性的客户端的源代码，如代码清单 14 所示。与方案四相比，这里在调用 KeyGenerator 的工厂方法时，传入序列键的名字作为参数；而在调用 getNextKey()方法时，则不需要传入序列键的名字作为参数。

代码清单 14：客户端的源代码

```
package com.javapatterns.keygen.ver5;
public class Client
{
    private static KeyGenerator keygen;
    public static void main(String[] args)
    {
        keygen = KeyGenerator.getInstance("PO_NUMBER");
        for (int i = 0 ; i < 20 ; i++)
        {
            System.out.println("key(" + (i+1)
                + ")= " + keygen.getNextKey());
        }
    }
}
```

可以看出，上面这个客户端对象首先创建一个多例类的实例，这个实例是以键名为内蕴状态的。然后就可以通过调用这个实例的 getNextKey()方法，得到所需的键值。这个系统的时序图如下图所示。



从图中可以看出，客户端首先通过调用多例类 KeyGenerator 的静态工厂方法 getInstance()得到一个多例类实例，与此同时 KeyGenerator 会创建一个 KeyInfo 对象。与方案四的情况不同的是，这里的静态工厂方法接受键名作为参量。

由于每一个多例对象都仅持有一个键名的 KeyInfo 对象，所以客户端可以向方案三一样调用 KeyGenerator 对象的 getNextKey()方法，得到所需的键值并返还给客户端。

运行的结果与上一个设计方案并无不同，所以不再重复。

18.4 讨 论

在上面给出的方案中，第四个和第五个方案都是具有实用价值的设计方案。读者可以尝试着将这两个方案应用到自己的设计中去，并根据具体的要求将设计方案进一步完善。

如果一个单例模式是一个聚集对象的话，那么这个聚集中所保存的是对其他对象的引用。一个多例模式则不同，多例对象使用一个聚集对象登记和保存自身的实例。由于这两种设计模式的相似之处，在很多情况下它们可以互换使用。本章所给出的设计方案四和设计方案五就是建立在单例聚集对象和多例对象的基础之上的、实现了相同功能的两种不同设计。

在方案四里面，KeyGenerator 对象是一个单例对象，同时也是一个聚集，而聚集中所存储的是 KeyInfo 对象。在方案五中，KeyGenerator 对象是一个多例对象，当然也是一个聚集对象，这个聚集中存储的是这个多例对象自身。

对于客户端来说，两种设计的区别并不大。使用方案四时，客户端创建一个单例对象，然后根据键名调用这个对象的 getNextKey()方法，从聚集中取出这个键名所对应的 KeyInfo 对象；如果聚集中没有这个 KeyInfo 对象，就创建一个新的对象，先将其存储到聚集中，然后返还给调用者。具体地讲，KeyInfo 会将键名和 KeyInfo 对象作为 HashMap 的键和对象存储在 HashMap 里面。

而当使用方案五时，客户端根据键名创建一个多例对象，这个对象以键名为内蕴状态。多例对象会将键名和自身的实例当做 HashMap 的键和对象存储在内部的 HashMap 对象里面。当客户端通过静态工厂方法请求 KeyGenerator 的实例时，会将所要求的键名传入；而在接到请求之后，KeyGenerator 首先会在自己的登记聚集中查找是否已经有这样一个满足要求的 KeyGenerator 对象。如果有，就将之提供给客户端；如果没有，就立即创建一个满足要求的 KeyGenerator 对象，将之登记到聚集里面，然后返还给客户端。

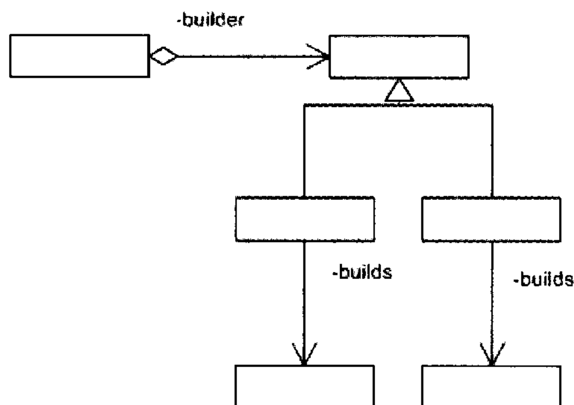
第 19 章　建造（Builder）模式

建造模式是对象的创建模式[GOF95]。建造模式可以将一个产品的内部表象与产品的生成过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。

19.1　引　言

产品的内部表象

一个产品常有不同的组成成分作为产品的零件，这些零件有可能是对象，也有可能不是对象，它们通常又叫做产品的内部表象（internal representation）。不同的产品可以有不同的内部表象，也就是不同的零件。使用建造模式可以使客户端不需要知道所生成的产品对象有哪些零件，每个产品的对应零件彼此有何不同，是怎么建造出来的，以及怎样组成产品。建造模式的简略类图如下图所示。



对象性质的建造

有些情况下，一个对象会有一些重要的性质，在它们没有恰当的值之前，对象不能作为一个完整的产品使用。比如，一个电子邮件有发件人地址、收件人地址、主题、内容、附录等部分，而在最起码的收件人地址得到赋值之前，这个电子邮件不能发出。

有些情况下，一个对象的一些性质必须按照某个顺序赋值才有意义。在某个性质没有赋值之前，另一个性质则无法赋值。这些情况使得性质本身的建造涉及到复杂的商业逻辑。

这时候，此对象相当于一个有待建造的产品，而对象的这些性质相当于产品的零件，

建造产品的过程是建造零件的过程。由于建造零件的过程很复杂，因此，这些零件的建造过程往往被“外部化”到另外一个称做建造者的对象里，建造者对象返还给客户端的是一个全部零件都建造完毕的产品对象。

建造模式非常适用于此种情况。建造模式利用一个导演者对象和具体建造者对象一个一个地建造出所有的零件，从而建造出完整的产品对象。建造者模式将产品的结构和产品的零件建造过程对客户端隐藏起来，把对建造过程进行指挥的责任和具体建造者零件的责任分割开来，达到责任划分和封装的目的。

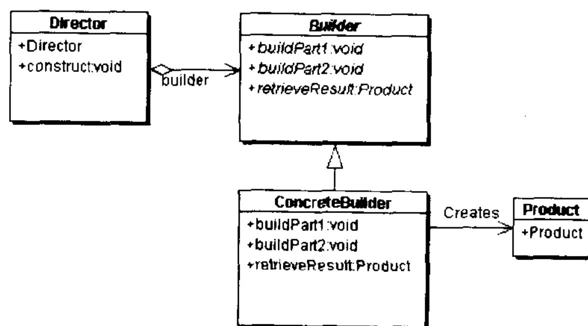
命名的考虑

应当指出的是，这些零件有可能是独立的对象，也有可能仅仅以对象的组成成分（如性质）的形式存在于产品对象内部。因此，本书使用“零件”，而不使用“零件对象”、“部件”和“构件”等往往意味着对象的词汇。对零件的产生，本书使用“建造”一词，而不使用“创建”或者“生成”一词，因为“创建”或者“生成”往往是针对对象而言的。

19.2 建造模式的结构

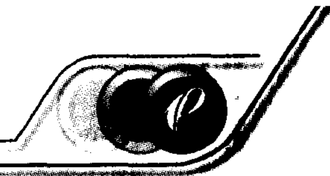
类图与角色

为方便读者了解建造模式的结构，本章给出一个示意性的实现，其类图如下图所示。



在这个示意性的系统里，最终产品 Product 只有两个零件，即 part1 和 part2。相应的建造方法也有两个：buildPart1()和 buildPart2()。同时可以看出本模式涉及到四个角色，它们分别是：

- 抽象建造者（Builder）角色：给出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此接口独立于应用程序的商业逻辑。模式中直接创建产品对象的是具体建造者（ConcreteBuilder）角色。具体建造者类必须实现这个接口所要求的两种方法：一种是建造方法，比如本例子中的 buildPart1()和 buildPar2()；另一种是结果返还方法，即本例子中的 retrieveResult()。一般来说，产品所包含的零件数目与建造方法的数目相符。换言之，有多少零件，就有多少相应的建造



方法。

- 具体建造者（Concrete Builder）角色：担任这个角色的是与应用程序紧密相关的一些类，它们在应用程序调用下创建产品的实例。这个角色要完成的任务包括：
  ①实现抽象建造者 Builder 所声明的接口，给出一步一步地完成创建产品实例的操作。
  ②在建造过程完成后，提供产品的实例。
- 导演者（Director）角色：担任这个角色的类调用具体建造者角色以创建产品对象。应当指出的是，导演者角色并没有产品类的具体知识，真正拥有产品类的具体知识的是具体建造者角色。
- 产品（Product）角色：产品（Product）便是建造中的复杂对象。一般来说，一个系统中会有多于一个的产品类，而且这些产品类并不一定有共同的接口，而完全可以是不相关联的。

导演者角色是与客户端打交道的角色。导演者角色将客户端创建产品的请求划分为对各个零件的建造请求，再将这些请求委派给具体建造者角色。具体建造者角色是做具体建造工作的，但是却不为客户端所知。

一般来说，每有一个产品类，就有一个相应的具体建造者类。这些产品应当有一样数目的零件，而每有一个零件就相应地在所有的建造者角色里有一个建造方法。

示意性类图中的建造模式只是一个最简单的例子，它包括了一个具体建造者类和一个产品类。在实际应用中，一般会有一个以上的具体建造者类和相应的一个以上的产品类。

源代码

下面给出了这个最简单的实现的示意性源代码。在将建造模式应用到自己的系统中时，读者可以参考后面的例子，并在自己的系统中把这里给出的源代码当做骨架，适当地添加上与自己的应用有关的部分。

首先，Director 类有一个建造方法（也就是 construct()方法），负责调用具体建造者对象的零件建造方法，即 buildPart1()、buildPart2()等方法，这里给出的源代码是一个简单的框架，如代码清单 1 所示。

代码清单 1：模式中 Director 类的源代码

```
package com.javapatterns.builder;
public class Director
{
    private Builder builder;
   /**
    *  产品构造方法，负责调用各个零件建造方法
    */
    public void construct()
    {
        builder = new ConcreteBuilder();
        builder.buildPart1();
```

```
            builder.buildPart2();
            builder.retrieveResult();
            //continue with other code
        }
}
```

抽象建造者通常与应用系统的商业逻辑无关，因此，可以用 Java 接口实现如代码清单 2 的源代码。

代码清单 2：抽象建造者 Builder 类的源代码

```
package com.javapatterns.builder;
abstract public class Builder
{
    /**
     *  产品零件建造方法
     */
    public abstract void buildPart1();
    /**
     *  产品零件建造方法
     */
    public abstract void buildPart2();
    /**
     *  产品返还方法
     */
    public abstract Product retrieveResult();
}
```

具体建造者角色是与应用系统的商业逻辑密切相关的，本书无法预测读者可能遇到的具体商业要求，因此这里仅仅给出一个骨架，如代码清单 3 所示。总的来说，这个角色应当将产品类实例化，并在各个建造方法中建造产品的各个零件，并通过方法 retrieveResult() 返回最终产品对象。

代码清单 3：具体建造者 ConcreteBuilder 类的源代码

```
package com.javapatterns.builder;
public class ConcreteBuilder extends Builder
{
    private Product product = new Product() ;
    /**
     *  产品返还方法
     */
    public Product retrieveResult()
    {
        return product;
    }
    /**
     *  产品零件建造方法
```

```
    */
    public void buildPart1()
    {
        //build the first part of the product
    }
   /**
    *  产品零件建造方法
    */
    public void buildPart2()
    {
        //build the second part of the product
    }
}
```

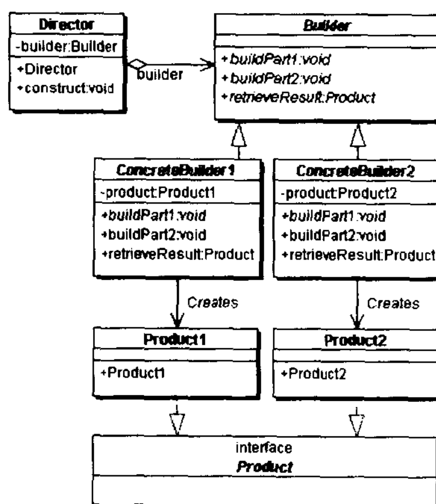
下面是产品类的源代码，如代码清单 4 所示。同样，这个类是与应用系统密切相关的，这里因此做了简化处理。

代码清单 4：产品类的示意性源代码

```
package com.javapatterns.builder;
public class Product
{
    // Anything pertaining to this product
}
```

多个产品类的情况

在上面的系统中只有一个产品类，对应地也只有一个具体建造者类。如果有两个产品类的话，就应当有两个具体建造者类。有两个产品类和具体建造者类的建造模式如下图所示。



产品类 Product1 和 Product2 各有两个零件，它们有不同的内部表象。在这里建造模式提供了一个对建造过程的封装，使得客户端不需要知道产品类的建造细节，便可以使用

一个建造接口生成具有不同的内部表象的 Product1 和 Product2 对象。

标识接口的应用

应当注意的是，retrieveProduct()方法是抽象建造者角色提供的，如果它返还的是 Product1 类型或者 Product2 类型的话，那么 ConcreteBuilder2 或者 ConcreteBuilder1 就会有问题，因为它们应当返还 Product2 和 Product1 类型。

解决的方案是为两个具体产品类提供一个共同的接口，形成它们共同的类型。由于建造模式的产品类往往是没有太多关系的一些类，因此，它们也不太可能有共同的接口。因此，这样使用一个标识接口为所有的具体产品类提供一个共同的类型就成为解决问题的方案，即所有的 retrieveProduct()方法都返还 Product 类型就可以了。

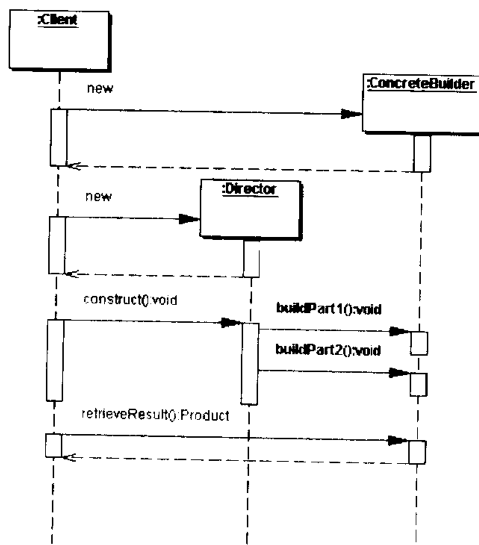
在上面的两个产品类和两个具体建造者类的类图里面，就给出了一个 Product 接口，作为 Product1 和 Product2 的共同接口。显然，Product 是一个标识接口（Marker Interface）模式的应用。

如果产品类是第三方提供的，不能修改，那么就只好将 retrieveProduct()方法从抽象建造者角色中去掉。这样具体建造者角色可以分别具有返回类型不同的 retrieveProduct()方法。如果 Director 对象需要调用这个方法，而变量的明显类型是 Builder，就必须首先将变量类型进行向下类型转换，使之成为具体建造者类型，然后调用相应的 retrieveProduct()方法。关于向下类型转换请参见本章后面所附的问答题。

必须指出的是，这种没有抽象产品类或者公共产品接口的情况更为普遍。

19.3 建造模式的活动序列

类图只能反映出模式的静态结构，对于创建模式而言，同样重要的还有系统的活动图。以前面介绍的示意性系统为例，建造者模式的活动序列可以用 UML 序列图表达，如下图所示。



客户端负责创建导演者和具体建造者对象。然后，客户把具体建造者对象交给导演者。客户一声令下，导演者操纵具体建造者，开始创建产品。具体建造者每接到导演者的一个指令，便按照指令向产品上增加一个构件。当产品完成后，建造者把产品返还给客户端。

虽然客户端确实是具体建造者对象，但是操纵具体建造者的任务却是属于导演者对象的。把创建具体建造者对象的任务交给客户端而不是导演者对象，是为了将导演者对象与具体建造者对象的耦合变成动态的，从而使导演者对象可以操纵数个具体建造者对象中的任何一个。

19.4　建造模式的实现

空的零件建造方法

阅读本章的读者，读到这里可能已经有了一些问题要问。其中一个常见的问题就是，当一个系统有多个产品类的时候，怎么能够保证它们所对应的具体建造者类都有同样的接口呢？

当然，产品的内部结构细节大多可以与具体建造者类的接口无关，因为接口仅仅给出建造方法的特征而已。但是接口确实规定了具体建造者类有几个零件建造方法，也就是说，抽象建造者角色确实规定了产品类必须有同样数目的零件，以及具体有几个零件。如果有一些产品有较多的零件，而有些产品有较少的零件，建造模式还可以使用吗？

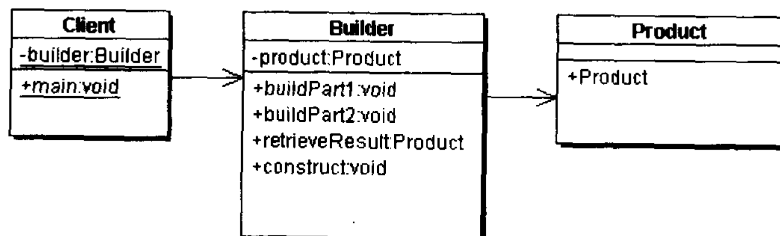
回答是肯定的。这些产品并不一定要有同样数目的零件才可以使用建造模式。如果一个产品有较少的零件，可以使用空的零件建造方法，忽略没有的零件。

这种为不感兴趣的方法提供平庸实现的做法是缺省适配（Default Adapter）模式的精神。

省略抽象建造者角色

建造模式在实现时可以根据具体情况做一些变化。

首先，如果设计师非常肯定系统只需要一个具体建造者角色的话，可以省略掉抽象建造者角色。抽象建造者角色存在的目的是规范具体建造者角色的行为，而系统如果只有一个具体建造者，那么这个规范者角色就不需要了，这时模式的类图如下图所示。



这时具体建造者类的源代码有所改变，下面给出这个类的源代码，如代码清单 5 所示。

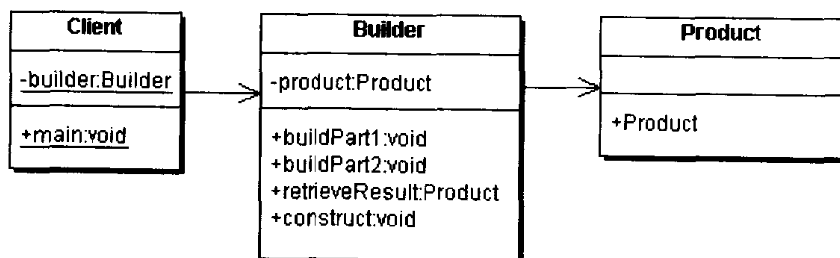
代码清单 5：具体建造者类的示意性源代码

```
package com.javapatterns.builder.simplified1;
abstract public class Director
{
    private ConcreteBuilder builder;
   /**
    *  产品构造方法，负责调用各个零件建造方法
    */
    public void construct()
    {
        builder.buildPart1();
        builder.buildPart2();
        Product product = builder.retrieveResult();
    }
}
```

其他的类没有改变。由于区别是静态的，因此，模式的活动时序图也没有明显的变化。

省略导演者角色

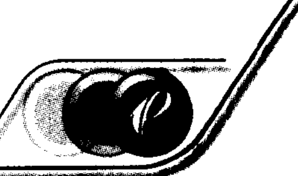
在具体建造者只有一个的情况下，如果抽象建造者角色已经被省略掉，那么还可以进一步省略掉导演者角色，这时的类图如下图所示。



可以看出，Builder 角色自己扮演了导演者和建造者的双重角色。此时有改变的类是 Builder 类，它的源代码如代码清单 6 所示。

代码清单 6：没有导演者角色的具体建造者类的示意性源代码

```
package com.javapatterns.builder.simplified2;
public class Builder
{
    private Product product = new Product();
   /**
    *  产品零件建造方法
    */
    public void buildPart1()
    {
        // Write your code here
```



```
    }
    /**
     *  产品零件建造方法
     */
    public void buildPart2()
    {
        // Write your code here
    }
    public Product retrieveResult()
    {
        return product;
    }
    /**
     *  产品构造方法，负责调用各个零件建造方法
     */
    public void construct()
    {
        buildPart1();
        buildPart2();
        Product product = retrieveResult();
    }
}
```

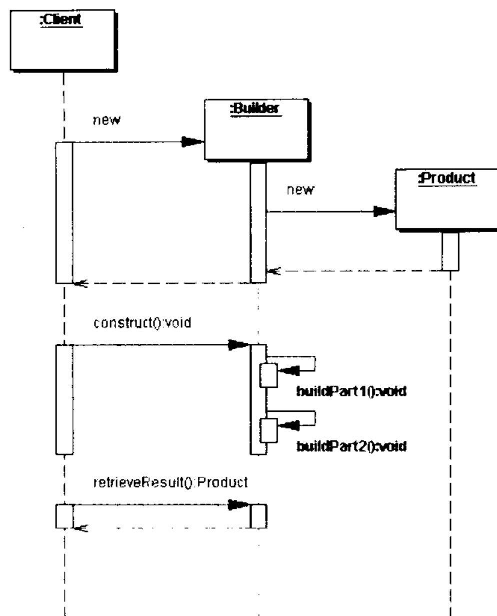
同样，客户端的源代码也需要做相应的调整，如代码清单 7 所示。

代码清单 7：一个示意性的客户端的源代码

```
package com.javapatterns.builder.simplified2;
public class Client
{
    private static Builder builder;
    static public void main(String[] args)
    {
        // 创建建造者对象
        builder = new Builder();
        // 调用建造者对象的产品构造方法
        builder.construct();
        // 调用建造者对象的产品返还方法以取得产品
        Product product = builder.retrieveResult();
    }
}
```

相应地，模式的时序图也有所不同，如下图所示。客户端创建一个 Builder 的实例，同时 Builder 创建一个 Product 的实例。客户端调用 construct()方法，此方法调用 buildPart1()和 buildPart2()等零件建造方法，建造出完整的产品对象。最后，客户端调用 retrieveProduct()方法得到最后产品的实例。

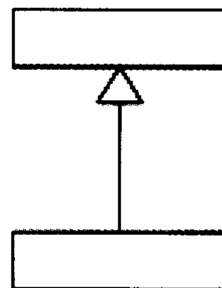




过渡到模版方法模式

准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现，这就是模版方法模式。

有意思的是，这个特殊退化的建造模式与模版方法模式有相似之处：construct()方法就相当于模版方法，这个方法调用其他的建造方法，如 buildPart1()、buildPart2()等基本方法，因此，这使得此系统与模版方法模式相同。模版方法模式的简略类图如右图所示。



这就是说，如果系统的要求发生变化、要求有不同的零件生成逻辑时，那么设计师就有两种选择：一是修改这个退化的建造模式，将它改回成为完全的建造模式，这当然就要涉及到代码的修改；二是不修改已有的代码，而是将 Builder 类扩展到不同的子类，在这些子类里面置换掉需要改变的建造方法。

第一种选择局限于建造模式；第二种选择则引入了模版方法（Template Method）模式。关于模版方法模式请见本书的“模版方法（Template Method）模式”一章。

合并建造者角色和产品角色

考虑建造模式失去抽象建造者角色和导演者角色之后，还可以进一步退化、从而失去具体建造者角色的情况。此时具体建造者角色与产品角色合并，从而使得产品自己就是自己的建造者。

显然，这样做混淆了对象的创立者和对象本身。但是，有时候一个产品对象有着固定的几个零件，而且永远只有这几个零件。此时将产品类与建造类合并，可以使系统简单易

读。可以看出，此时这些零件就变成了产品的性质。

读者可以在后面看到，JavaMail 库包中的 Message 类就是一个退化的建造模式，它的性质如 from、recipient、subject、text 等，均可以看做是 Message 的“零件”。

在很多情况下，建造模式实际上是将一个对象的性质建造过程外部化到独立的建造者对象中，并通过一个导演者角色对这些外部化的性质赋值过程进行协调。作为理解建造模式的办法，这是很有启发意义的。

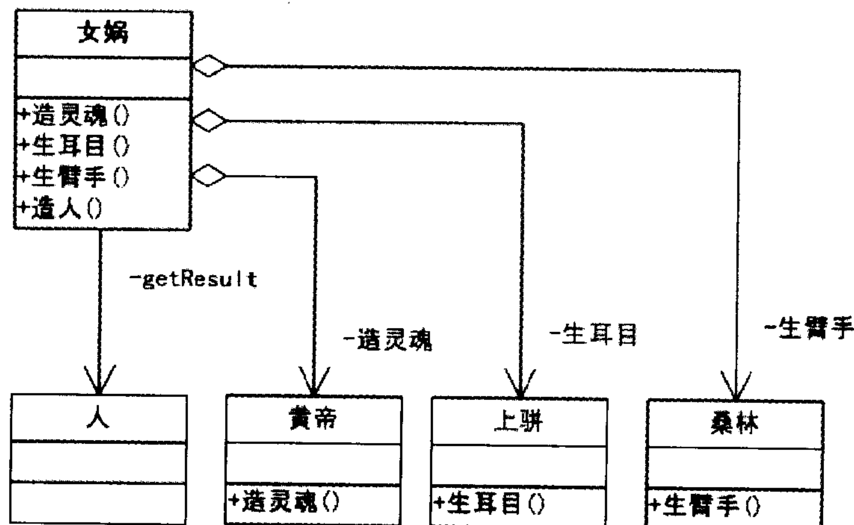
建造者角色可以有多个产品构造方法

建造者角色可以同时有多于一个的产品构造方法，可以重载或者使用不同的方法名。这样可以使得产品对象的创建方式多样化。

19.5　众神造人的神话故事

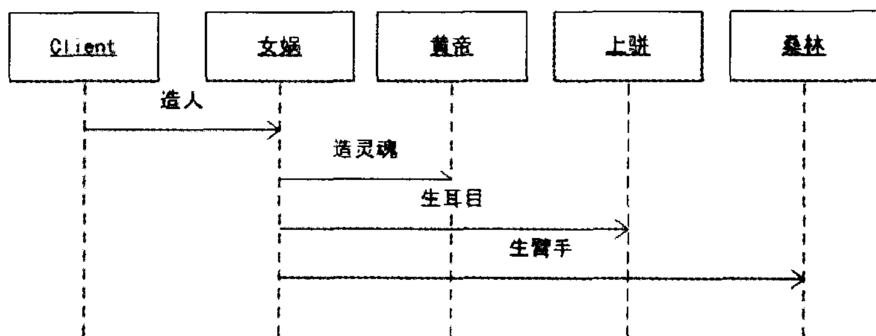
《淮南子·说林篇》说：“黄帝生阴阳，上骈生耳目，桑林生臂手，此女娲所以七十化也”，这就是女娲与诸神共同造人之说。它的关键之处是把人划分成灵魂（阴阳）、感觉器官（耳目）和四肢（臂手）等 70 个内部表象，而女娲则在一个过程中一部分一部分地通过向诸神的委派创造出最终产品类的所有组成部分，并组成最终产品——“人”。

在这个诸神造人的系统中，女娲本人扮演建造者（Builder）的角色，她控制着造人的诸神：黄帝、上骈和桑林。与女娲抟土造人和举绳造人的神话不同，众神造人的神话是说人是一部分一部分地造出来的。黄帝创造了人的灵魂，上骈造出人的感官，桑林造出人的手臂。使用 UML 类图描述众神造人的传说，如下图所示。



在上面的类图中，“女娲”扮演一个建造者角色，它将建造人的任务委派给诸神。对于黄帝来说，被委派的就是“造灵魂”的操作，对于上骈来说，得到的是“生耳目”的任务，而桑林得到的是“生臂手”的任务等。

“灵魂”加上“耳目”和“臂手”等就组成一个“人”，其UML类图如下图所示。



女娲的“造人”操作分别调用黄帝的“造灵魂”操作、上骈的“生耳目”操作和桑林的“生臂手”操作等，并最后调用 getResult()方法，得到造好的一个“人”，作为最终产品。

19.6 JavaMail 中的建造模式

本书所有关于 JavaMail 的讨论都是基于 JavaMail 1.2 或更新的版本的。如果读者没有这个版本，请首先从 java.sun.com 网站下载最新版的 JavaMail 库包，再试图运行本章提供的例子。

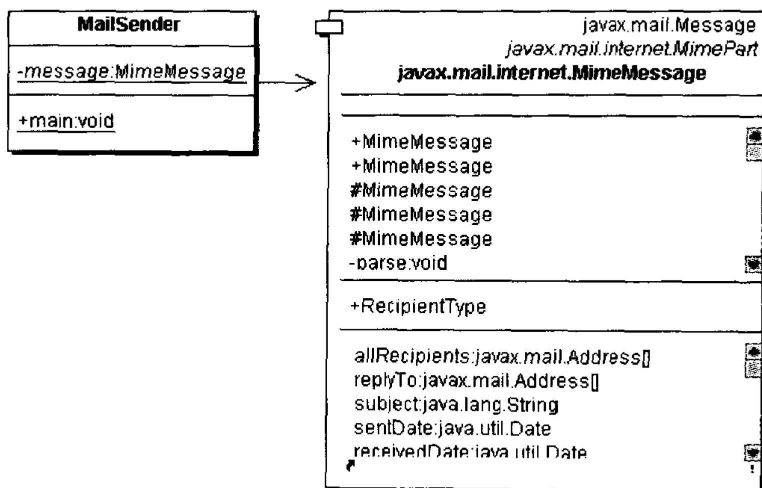
什么是 JavaMail

JavaMail 是一组 J2SE 的扩展 API 的一个类库。使用 JavaMail，设计师可以很容易地开发出功能齐备的客户端电子邮件软件。JavaMail 支持多个协议，如 IMAP4、POP3 以及 SMTP 等，并易于适应将来可能出现的新的协议。JavaMail 利用了 Java Activation Framework（JAF）库包，因此，读者在下载 JavaMail 库包的同时还需下载 JAF 库包。JAF 库包提供了统一的处理现有资料格式的方式。

JavaMail 大量地使用抽象工厂模式和其他的设计模式，包括建造模式。

建造模式的应用

JavaMail 中的 Message 和 MimeMessage 等类均是退化的建造模式的应用。在这里，产品类自己提供建造方法。客户端可以通过直接调用这些建造方法一步一步地建造出完整的产品对象，在这里便是 Message 和 MimeMessage 等对象。JavaMail 的使用如下图所示。



作为客户端和导演对象，MailSender 通过一步步地提供 MimeMessage 对象的“零件”性质，也就是 from、recipient、subject、text 等，最后得到整个的产品对象，也就是 MimeMessage 对象。MailSender 的源代码如代码清单 8 所示。

代码清单 8：一个简单的发送电子邮件的客户端 MailSender 的源代码

```
package com.javapatterns.builder.javamail;
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
public class MailSender
{
    private static MimeMessage message;
    public static void main(String[] args)
    {
        //你的 SMTP 服务器地址
        String smtpHost="smtp.mycompany.com";
        //发送者的地址
        String from="jeff.yan@mycompany.com";
        //收信者的地址
        String to ="ni.hao@youcompany.com";

        Properties props = new Properties();
        props.put("mail.smtp.host", smtpHost);

        Session session = Session.getDefaultInstance(props, null);

        try
        {
            InternetAddress[] address = {new InternetAddress(to)};
            // 创建 message 对象
```

```
                message = new MimeMessage(session);
                // 建造发件人位元址零件
                message.setFrom(new InternetAddress(from));
                // 建造收件人位元址零件
                message.setRecipients(Message.RecipientType.TO, address);
                // 建造主题零件
                message.setSubject("Hello from Jeff");
                // 建造发送时间零件
                message.setSentDate(new Date());
                // 建造内容零件
                message.setText("Hello,\nHow are things going?");

                // 发送邮件，相当于产品返还方法
                Transport.send(message);
                System.out.println("email has been sent.");
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
```

读者可以通过将这个例子与下面的例子比较，体会到本章所提出的建造模式的“性质外部化”的描述。

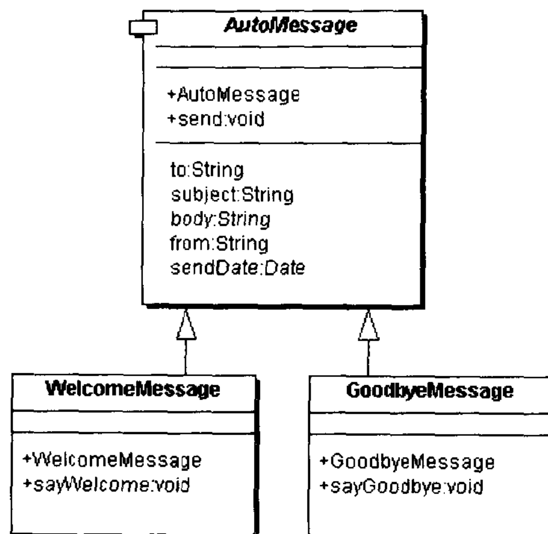
19.7 一个发送邮件的例子

本节给出一个例子，以说明建造模式的应用。

假设有一个使用 Java 语言建造的电子杂志系统，定期地向用户的电子邮件信箱发送电子杂志。用户可以通过网页订阅电子杂志，也可以通过网页结束订阅。当客户开始订阅时，系统发送一个电子邮件表示欢迎，当客户结束订阅时，系统发送一个电子邮件表示欢送。本例子就是这个系统负责发送“欢迎”和“欢送”邮件的模块。

产品及其零件

在本例子里，产品类就是发给某个客户的“欢迎”和“欢送”邮件，如下图所示。



应当指出的是，虽然在这个例子里面各个产品类均有一个共同的接口，但这仅仅是本例子特有的，并不代表建造模式的特点。建造模式可以应用到具有完全不同接口的产品类上。

下面就是抽象类 AutoMessage 的源代码，如代码清单 9 所示。可以看出，send()操作仅仅是示意性的，并没有给出任何发送电子邮件的代码，因为本书希望能将读者的注意力集中在与模式有关的方面。

代码清单 9：抽象类 Message 的源代码

```
package com.javapatterns.builder.messagebuilder;
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
abstract public class AutoMessage
{
    protected String subject = "";
    protected String body = "";
    protected String from = "";
    protected String to = "";
    protected Date sendDate = null;
    public AutoMessage()
    {
    }
    public void send()
    {
        //你的 SMTP 服务器地址
        String smtpHost="smtp.mycompany.com";
        //发送者的地址
        String from="jeff.yan@mycompany.com";
        //收信者的地址
```

```
        String to ="ni.hao@yourcompany.com";

        Properties props = new Properties();
        //The protocol to use is SMTP
        props.put("mail.smtp.host", smtpHost);

        Session session = Session.getDefaultInstance(props, null);

        try
        {
            InternetAddress[] address = {new InternetAddress(to)};
            MimeMessage message;
            // 创建 message 对象
            message = new MimeMessage(session);
            // 建造发件人位元址零件
            message.setFrom(new InternetAddress(from));

            // 建造收件人位元址零件
            message.setRecipients(Message.RecipientType.TO, to);
            // 建造主题零件
            message.setSubject("Hello from Jeff");

            // 建造发送时间零件
            message.setSentDate(sendDate);

            // 建造内容零件
            message.setText("Hello,\nHow are things going?");

            // 发送邮件，相当于产品接收方法
            Transport.send(message);
            System.out.println("email has been sent.");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
    /**
     *  收件人地址的取值方法
     */
    public String getTo()
    {
        return to;
    }
    /**
     *  收件人地址的赋值方法
```

```
    */
    public void setTo(String to)
    {
        this.to = to;
    }
   /**
    *  邮件主题的取值方法
    */
    public String getSubject()
    {
        return subject;
    }
   /**
    *  邮件主题的赋值方法
    */
    public void setSubject(String subject)
    {
        this.subject = subject;
    }
   /**
    *  邮件内容的取值方法
    */
    public String getBody()
    {
        return body;
    }
   /**
    *  邮件内容的赋值方法
    */
    public void setBody(String body)
    {
        this.body = body;
    }
   /**
    *  发件人地址的取值方法
    */
    public String getFrom()
    {
        return from;
    }
   /**
    *  发件人地址的赋值方法
    */
    public void setFrom(String from)
    {
        this.from = from;
```

```
    }
    /**
     *  发送时间的取值方法
     */
    public Date getSendDate()
    {
        return sendDate;
    }
    /**
     *  发送时间的赋值方法
     */
    public void setSendDate(Date sendDate)
    {
        this.sendDate = sendDate;
    }
}
```

下面是具体产品类 WelcomeMessage 的源代码，如代码清单 10 所示。

代码清单 10：WelcomeMessage 类的源代码

```
package com.javapatterns.builder.messagebuilder;
public class WelcomeMessage extends AutoMessage
{
    /**
     *  构造子
     */
    public WelcomeMessage()
    {
        System.out.println(" Entering Welcome Message. ");
    }
    /**
     *  某个商业方法
     */
    public void sayWelcome()
    {
        System.out.println(" Welcome. ");
    }
}
```

下面是具体产品类 GoodbyeMessage 的源代码，如代码清单 11 所示。

代码清单 11：GoodbyeMessage 类的源代码

```
package com.javapatterns.builder.messagebuilder;
public class GoodbyeMessage extends AutoMessage
{
    /**
     *  构造子
```

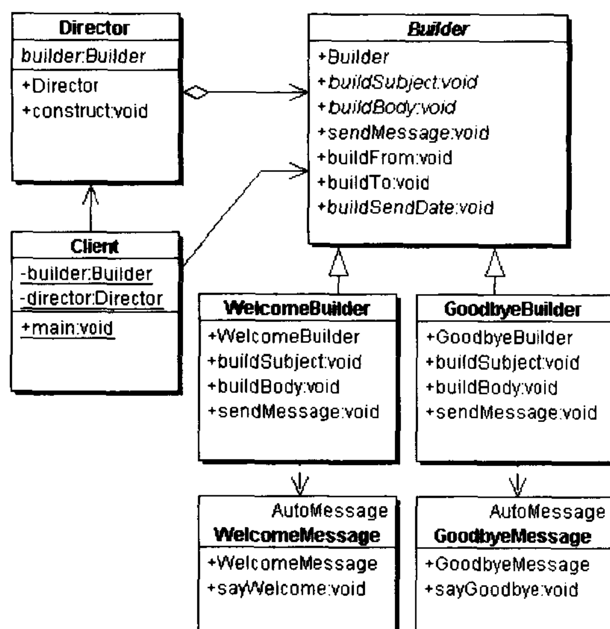
```
    */
    public GoodbyeMessage()
    {
        System.out.println(" Entering Goodbye Message. ");
    }
   /**
    *  某个商业方法
    */
    public void sayGoodbye()
    {
        System.out.println(" Goodbye. ");
    }
}
```

由于电子邮件可以划分成发件人地址、收件人地址、主题、内容等部分（或称零件），因此，电子邮件建造过程可以划分成为几个零件的建造过程，包括 buildTo()、buildFrom()、buildSendDate()、buildSubject()以及 buildBody()。

读者应当看出，邮件是最终产品，而 To、From、SendDate 和 Subject 以及邮件内容则是产品零件。如果再有一个导演者类，不正是一个标准的建造模式吗？确实，这正是建造模式的应用。

系统的设计

下图所示就是这个系统的类图。读者可以看出，类图带有建造模式的显著特点。



这个系统含有客户端（Client）、导演角色（Director）、抽象建造者（Builder）、具体建造者（WelcomeBuilder 和 GoodbyeBuilder）、产品（WelcomeMessage 和 GoodbyeMessage）等角色。

下面是导演者角色的源代码，如代码清单 12 所示。这个类提供一个 construct()方法，此方法调用建造对象的建造方法，包括 buildTo()、buildFrom()、buildSubject()、buildBody()、buildSendDate()等，从而一部分一部分地建造出产品对象，即 AutoMessage 对象。

代码清单 12：导演类 Director 的源代码

```
package com.javapatterns.builder.messagebuilder;
public class Director
{
    Builder builder;
   /**
    *  构造子
    */
    public Director(Builder builder)
    {
        this.builder = builder;
    }
   /**
    *  产品构造方法，负责调用各零件建造方法
    */
    public void construct(String toAddress,
        String fromAddress)
    {
        this.builder.buildSubject();
        this.builder.buildBody();
        this.builder.buildTo( toAddress );
        this.builder.buildFrom( fromAddress );
        this.builder.buildSendDate();
        this.builder.sendMessage();
    }
}
```

抽象建造者类规范出所有的具体建造者类需要实现的接口，如代码清单 13 所示。

代码清单 13：抽象建造者类 Builder 的源代码

```
package com.javapatterns.builder.messagebuilder;
import java.util.Date;
abstract public class Builder
{
    protected AutoMessage msg ;
   /**
    *  构造子
    */
    public Builder()
    {
    }
   /**
```

```
    *  主题零件的建造方法
    */
    public abstract void buildSubject();
   /**
    *  内容零件的建造方法
    */
    public abstract void buildBody();

   /**
    *  发件人零件的建造方法
    */
    public void buildFrom(String from)
    {
        msg.setFrom( from );
    }
   /**
    *  收件人零件的建造方法
    */
    public void buildTo(String to)
    {
        System.out.println(to);
        msg.setTo( to );
    }
   /**
    *  发送时间零件的建造方法
    */
    public void buildSendDate()
    {
        msg.setSendDate(new Date());
    }
   /**
    *  邮件产品完成后，用此方法发送邮件
    *  此方法相当于产品返还方法
    */
    public void sendMessage()
    {
        msg.send();
    }
}
```

具体建造者类 WelcomeBuilder 实现抽象建造者角色规定出的接口，主要是 buildSubject()和 buildBody()两个方法，如代码清单 14 所示。

代码清单 14：具体建造者类 WelcomeBuilder 的源代码

```
package com.javapatterns.builder.messagebuilder;
import java.util.Date;
public class WelcomeBuilder extends Builder
```

```
{
    private static final String subject =
        "Welcome to philharmony news group!";
    /**
     * 构造子
     */
    public WelcomeBuilder()
    {
        msg = new WelcomeMessage();
    }
    /**
     * 主题零件的建造方法
     */
    public void buildSubject()
    {
        msg.setSubject( subject );
    }
    /**
     * 内容零件的建造方法
     */
    public void buildBody()
    {
        String body =
            "Congratulations for making the right choice!";
        msg.setBody(body);
    }
    /**
     * 邮件产品建造完成后，发送邮件
     * 此方法相当于产品返还方法
     */
    public void sendMessage()
    {
        msg.send();
    }
}
```

GoodbyeBuilder 是另一个具体建造者，它实现了抽象建造者的接口，如代码清单 15 所示。

代码清单 15：具体建造者类 GoodbyeBuilder 的源代码

```
package com.javapatterns.builder.messagebuilder;
import java.util.Date;
public class GoodbyeBuilder extends Builder
{
    private static final String subject =
        "Thank you for being with us!";
    /**
```

```
    *  构造子
    */
    public GoodbyeBuilder()
    {
        msg = new GoodbyeMessage();
    }
   /**
    *  主题零件的建造方法
    */
    public void buildSubject()
    {
        msg.setSubject( subject );
    }
   /**
    *  内容零件的建造方法
    */
    public void buildBody()
    {
        String body = "Oops! You have chosen to leave.";
        msg.setBody( body );
    }
   /**
    *  邮件产品建造完成后，发送邮件
    */
    public void sendMessage()
    {
        msg.send();
    }
}
```

具体建造者类 GoodbyeBuilder 实现了抽象建造者角色规定出的接口，其中包括了 buildSubject()和 buildBody()等方法。Client 类的源代码如代码清单 16 所示。

代码清单 16：代表客户端的 Client 类的源代码

```
package com.javapatterns.builder.messagebuilder;
public class Client
{
    private static Builder builder;
    private static Director director;
    static public void main(String[] args)
    {
        builder = new WelcomeBuilder();

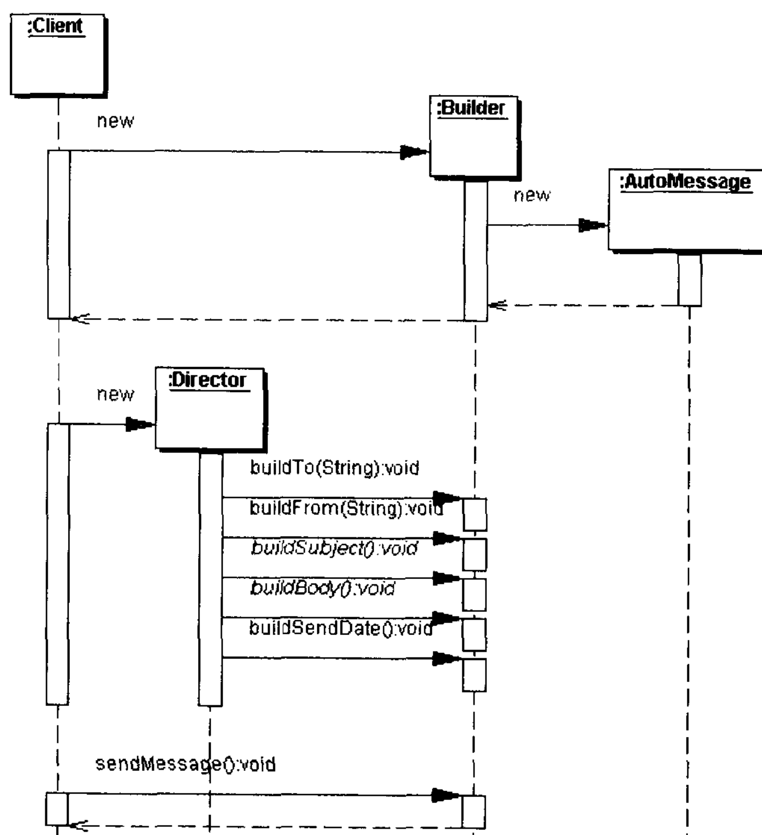
        director = new Director(builder);
        director.construct( "jeffyan77@yahoo.com",
            "javapatterns@yahoo.com" );
    }
```

}

可以看出，Client 类首先创建了 Builder 对象，然后创建了 Director 对象。在创建 Director 对象时，把 Builder 对象当做参数传入到 Director 类的构造子里。

系统的时序图

客户端负责创建导演者和具体建造者对象。然后，客户把具体建造者对象交给导演者。客户一声令下，导演者对象操纵具体建造者对象（在这里便是 WelcomeBuilder），开始创建产品对象。具体建造者每次接到导演者的指令后，便按照指令建造发信人位元址、收信人地址、标题、信体和发送日期等。当邮件建造完成后，建造者对象把邮件作为产品返还给客户端对象。系统的时序图如下图所示。



19.8 在什么情况下使用建造模式

在以下情况下应当使用建造模式：

（1）需要生成的产品对象有复杂的内部结构。每一个内部成分本身可以是对象，也可以仅仅是一个对象（即产品对象）的一个组成成分。

（2）需要生成的产品对象的属性相互依赖。建造模式可以强制实行一种分步骤进行的建造过程，因此，如果产品对象的一个属性必须在另一个属性被赋值之后才可以被赋值，

使用建造模式便是一个很好的设计思想。

比如，在美国的各个州购买商品需要缴纳购买人所在州的销售税，而各个州的销售税均不相同。因此，当消费者打电话购买商品时，一个销售系统必须知道消费者是从哪一个州付账，才可以决定销售税的比例。对应的信息系统必须要求使用者将顾客所在的州输入到系统中，才能看到销售税和总额。建造模式适合于在这种情况下使用。

有时产品对象的属性并无彼此依赖的关系，但是在产品的属性没有确定之前，产品对象不能使用。这时产品对象的实例化，属性的赋值和使用仍然是分步骤进行的。因此，建造模式仍然有意义。

（3）在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到。

同时，使用建造模式主要有以下的效果：

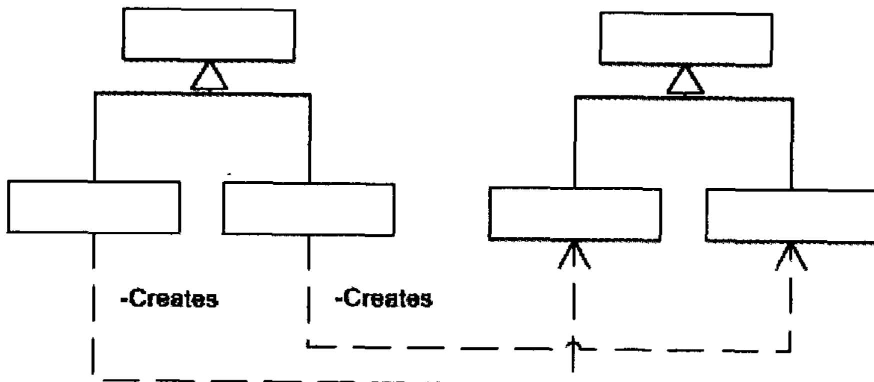
- 建造模式的使用使得产品的内部表象可以独立地变化。使用建造模式可以使客户端不必知道产品内部组成的细节。
- 每一个 Builder 都相对独立，而与其他的 Builder 无关。
- 模式所建造的最终产品更易于控制。

19.9 建造模式与其他模式的关系

建造模式与抽象工厂模式和策略模式等有相似之处，但又有不同的地方。

建造模式与抽象工厂模式的区别

读者可能已经注意到了，建造模式与抽象工厂模式非常相像，而两者又都是用来创建同时属于几个产品族的对象的模式。那么这两种模式有什么样的区别呢？抽象工厂模式的简略类图如下图所示。



在抽象工厂模式中，每一次工厂对象被调用时都会返还一个完整的产品对象，而客户端有可能会决定把这些产品组装成一个更大更复杂的产品，也有可能不会。建造类则不同，它一点一点地建造出一个复杂的产品，而这个产品的组装过程就发生在建造者角色内部。

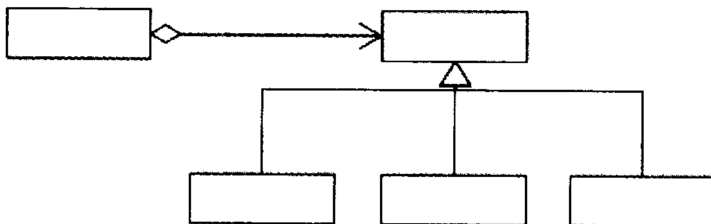
建造者模式的客户端拿到的是一个完整的最后产品。

换言之，虽然抽象工厂模式与建造模式都是设计模式，但是抽象工厂模式处在更加具体的尺度上，而建造模式则处于更加宏观的尺度上。一个系统可以由一个建造模式和一个抽象工厂模式组成，客户端通过调用这个建造角色，间接地调用另一个抽象工厂模式的工厂角色。工厂模式返还不同产品族的零件，而建造者模式则把它们组装起来。

比如仍以众神造人为例，女娲利用建造模式负责把灵魂、耳目、手臂等组合成一个完整的人，而黄帝、上骈、桑林各自利用工厂模式创造出灵魂、耳目、手臂等。女娲不必考虑灵魂、耳目、手臂是什么样子、怎么创造出来的，这就成为一个由建造模式和抽象工厂模式组合而成的系统。

建造模式与策略模式的区别

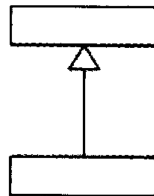
建造模式在结构上很接近于策略模式，事实上建造模式是策略模式的一种特殊情况，这两种模式的区别在于它们的用意不同。建造模式适用于为客户端一点一点地建造新的对象，而不同类型的具体建造者角色虽然都拥有相同的接口，但是它们所创建出来的对象则可能完全不同。策略模式的简略类图如下图所示。



策略模式的目的是为算法提供抽象的接口。换言之，一个具体策略类把一个算法包装到一个对象里面，而不同的具体策略对象为一种一般性的服务提供不同的实现。

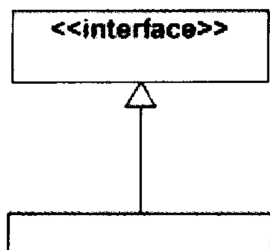
从建造模式过渡到模版方法模式

正如前面所指出的，建造模式在退化、失去导演者角色后，可以顺理成章地发展到模版方法模式。模版方法模式的简略类图如右图所示。



使用缺省适配模式

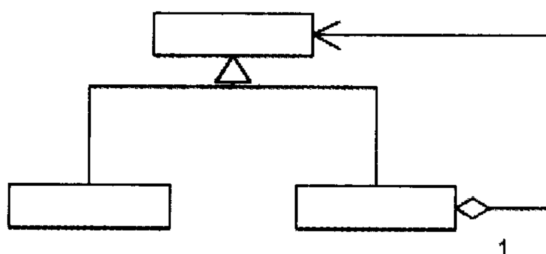
如果某个产品没有某个零件的话，那么此产品所对应的具体建造者类便无法实现对应于那个零件的建造方法。这时便可以使用缺省适配模式，提供一个建造方法的平庸实现，也就是不产生真正的建造行为的实现。缺省适配模式的简略类图如下图所示。



建造模式与合成模式的关系

正如本章前面所说的，产品的零件可以是对象，也可以不是对象，而是对象的某种组成成分。当产品的零件确实是对象时，产品对象就变成了复合对象，因为产品内部还含有子对象。这种对象内含有子对象的结构，可以使用合成模式描述。

换言之，合成模式描述一个对象树的组成结构，而建造模式则可以用来描述对象树的生成过程。合成模式的简略类图如下图所示。



问答题

1. 请给出有两个产品类和具体建造者类的建造模式类图中各个角色的源代码。
2. 请给出一个有三个产品类的建造模式的例子。
3. 下面是古时女子写给心上人的情诗："一二三四五六七八九十"。请问这是什么意思，蕴含什么模式。
4. 请解释什么是向上类型转换（upcast）。
5. 请解释什么是向下类型转换（downcast）。

问答题答案

1. 这个系统所涉及的各个角色的源代码如下所示。首先是抽象建造者角色的源代码，如代码清单 17 所示。

代码清单 17：抽象建造者角色的源代码

```
package com.javapatterns.builder.extended;
abstract public class Builder
```

```
{
    /**
     *  产品零件建造方法
     */
     public abstract void buildPart1();
    /**
     *  产品零件建造方法
     */
     public abstract void buildPart2();
    /**
     *  产品返还方法
     */
     public abstract Product retrieveResult();
}
```

可以看出，上面的抽象建造者角色规定了两个零件的建造方法，分别对应于产品的两个零件，以及一个产品返还方法。

下面是具体建造者角色 ConcreteBuilder1 的源代码，如代码清单 18 所示。这个建造者类实现了抽象建造者所规定的两个建造方法和一个零件返还方法。

代码清单 18：第一个建造者类的源代码

```
package com.javapatterns.builder.extended;
public class ConcreteBuilder1 extends Builder
{
     private Product1 product = new Product1() ;
    /**
     *  产品零件建造方法
     */
     public void buildPart1()
     {
         //build the first part of the product
     }
    /**
     *  产品零件建造方法
     */
     public void buildPart2()
     {
         //build the second part of the product
     }
    /**
     *  产品返还方法
     */
     public Product retrieveResult()
     {
         return product;
     }
```



```
}
```

ConcreteBuilder2 的源代码与上面的源代码基本类似，如代码清单 19 所示。

代码清单 19：第二个具体建造者类的源代码

```
package com.javapatterns.builder.extended;
public class ConcreteBuilder2 extends Builder
{
    private Product2 product = new Product2() ;
   /**
    *   产品零件建造方法
    */
    public void buildPart1()
    {
        //build the first part of the product
    }
   /**
    *   产品零件建造方法
    */
    public void buildPart2()
    {
        //build the second part of the product
    }
   /**
    *   产品返还方法
    */
    public Product retrieveResult()
    {
        return product;
    }
}
```

产品类 Product1 的源代码如代码清单 20 所示。

代码清单 20：第一个产品类的源代码

```
package com.javapatterns.builder.extended;
public class Product1
{
   /**
    *   构造子
    */
    public Product1()
    {
        //Write your code here
    }
}
```

产品类 Product2 的源代码如代码清单 21 所示。

代码清单 21：第二个产品类的源代码

```
package com.javapatterns.builder.extended;
public class Product2
{
    /**
     *  构造子
     */
    public Product2()
    {
        //Write your code here
    }
}
```

导演者角色提供一个建造方法，客户端可以通过调用这个建造方法操控各个建造者对象，如代码清单 22 所示。

代码清单 22：导演角色的源代码

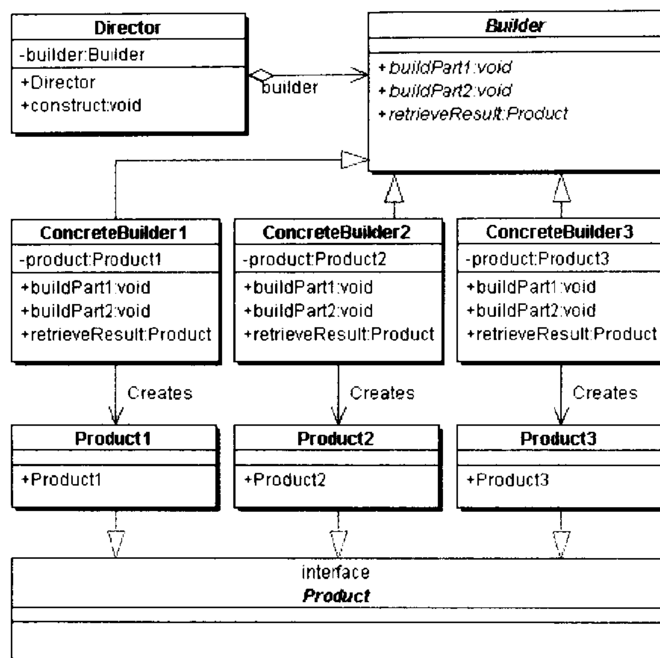
```
package com.javapatterns.builder.extended;
public class Director
{
    private Builder builder;
    /**
     *  构造子
     */
    public Director(Builder builder)
    {
        this.builder = builder;
    }
    /**
     *  产品构造方法，负责调用个零件建造方法
     */
    public void construct()
    {
        builder = new ConcreteBuilder();
        builder.buildPart1();
        builder.buildPart2();
        builder.retrieveResult();
        //continue with other code
    }
}
```

上面给出了一个两个产品类共同的接口，也就是一个标识接口，如代码清单 23 所示。

代码清单 23：标识接口 Product 的源代码

```
package com.javapatterns.builder.extended;
public interface Product{ }
```

2. 有三个产品类的建造模式的类图如下图所示。



每一个产品类都有相应的一个具体建造者类与之对应，因此，在结构图中出现了三个具体建造者类。

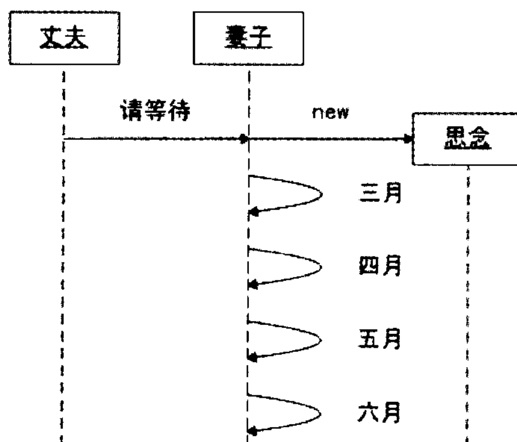
上面使用了一个标识接口 Product 作为三个产品类的共同类型。

3. 这是一首古诗："一别之后，两地相思，三月桃花随水转，四月琵琶未黄，奴我欲对镜心却乱。五月石榴红似火，偏遇冷雨浇花端。六月伏天人人摇扇独我心寒。七弦琴无心弹，八行书无处传，九连环从中折断，十里长庭望眼欲穿。"

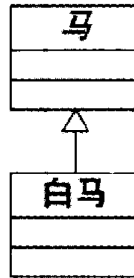
这是一个建造模式的应用。

显然，古诗描述的是一个数着月份逐步完成几个月思念的过程。丈夫请求妻子等待他回来，因此丈夫相当于建造模式中的导演角色。妻子接到丈夫的请求便开始一步步地建造"思念"产品，妻子扮演具体建造者的角色。

如果使用面向对象的系统模拟这个古代的妻子等待丈夫归来的过程的话，系统的时序图如下图所示。



4. 从子类向基类的类型转换叫做向上类型转换（upcast）。以白马和马的例子来说，“马”是“白马”的基类，“白马”是“马”的子类，如下图所示。



下面给出的代码涉及到从“白马”类型向“马”类型的转换，这就是“向上类型转换”，如代码清单 24 所示。

代码清单 24：涉及到向上类型转换的源代码

```
白马 x = new 白马（）;

马 y = x;
```

向上类型转换永远是安全和成功的。

5. 向下类型转换（downcast）是从基类到子类的类型转换。仍然以白马和马的例子来说，将一个静态类型为“马”的变量转换为“白马”类型，这就是向下类型转换。

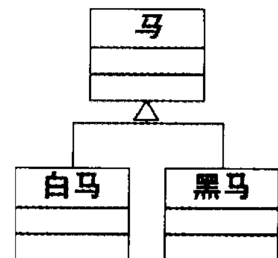
在下面的源代码中涉及到一个向下类型转换，也就是将静态类型为“马”的变量 *x* 转换成为“白马”类型，如代码清单 25 所示。

代码清单 25：涉及到向下类型转换的源代码

```
马 x = new 白马（）;

白马 y = （白马） x;
```

向下类型转换涉及到 Java 语言的运行时期的类型检查（Run-Time Type Identification），当检查不能通过时，向下类型转换就会失败，并抛出 ClassCastException 异常。



比如考察下面的黑马和白马的情况，白马、黑马与马的关系如右图所示。

那么下面的源代码所涉及的从马向白马的向下类型转换就会失败，并抛出一个 ClassCastException 异常，如代码清单 26 所示。

代码清单 26：一个失败的向下类型转换

```
马 x = new 黑马（）;
白马 y = （白马） x;
```

第 20 章　原始模型（Prototype）模式

原始模型模式属于对象的创建模式[GOF95]。通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。这就是原始模型模式的用意。

20.1　引　　言

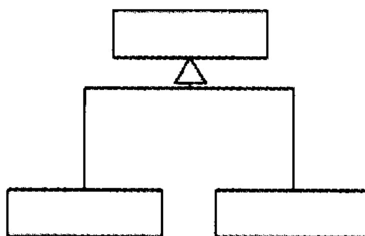
从孙大圣的手段谈起

孙悟空在与黄风怪的战斗中，"使一个身外身的手段：把毫毛揪下一把，用口嚼得粉碎，望上一喷，叫声'变'，变有百十个行者，都是一样打扮，各执一根铁棒，把那怪围在空中。"换言之，孙悟空可以根据自己的形象，复制出很多"身外之身"来。

老孙的这种身外身的手段在面向对象的设计领域里叫做原始模型（Prototype）模式。

原始模型模式

Java 语言的构件模型直接支持原始模型模式。所有的 JavaBean 都继承自 java.lang.Object，而 Object 类提供一个 clone()方法，可以将一个 JavaBean 对象复制一份。但是，这个 JavaBean 必须实现一个标识接口 Cloneable，表明这个 JavaBean 支持复制。如果一个对象没有实现这个接口而调用 clone()方法，Java 编译器会抛出 CloneNotSupportedException 异常。原始模型模式的简略类图如下图所示。（关于标识接口的概念及相关讨论，请见本书的"专题：Java 接口"一章）



下面来复习一下 Java 语言中复制对象的办法，就从变量、对象和对象的引用谈起吧。

20.2 变量、对象以及对象的引用

在 Java 语言中，对象就是类的实例。在一般情况下，当把一个类实例化时，此类的所有的成员，包括变量和方法，都被复制到属于此数据类型的一个新的实例中去。比如下面是产生一个数据类型是 Panda（熊猫）类的，叫做 myPanda 的新对象的办法：

```
Panda myPanda = new Panda();
```

上面的语句做了如下的事情：

- 创建了一个 Panda 类型的变量，称为 myPanda。
- 建立了一个 Panda 类的对象。
- 使变量 myPanda 指到这个新的对象。

如果将上面的语句分成两个步骤：

```
Panda myPanda;
myPanda = new Panda();
```

可以清楚地看到，在第一行建立了一个变量，称做 myPanda，可以指到 Panda 类对象上面。但在第一行结束时并没有指到它上面，只是在第二行才真正指到这样的一个对象上。事实上，如果省略第二行而直接使用 myPanda 变量的话，会产生“变量没有初始化”的运行时期错误。

现在回到上面的第二行，这一行语句创建了一个对象。当对象被创建时不是 myPanda 对象，而是一个无名的对象。随后，这个无名的对象马上就有了一个名字，也就是 myPanda。那么会不会在一个对象被创建出来后，就没有名字，而永远保持无名状态呢？请看下面的例子：

```
myPanda.eat(new Bamboo("green"));
```

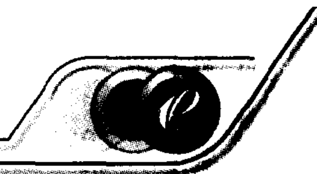
在上面的例子中，new Bamboo("green")创建了一个无名的 Bamboo（竹子）类的对象。显然，Bamboo 类的构造子有一个参量，就是颜色，上面特地选了绿色。接收这个无名对象的是 myPanda 对象的 eat()方法，而 eat()方法显然并不在意它所接收的对象有没有名字。在 eat()使用此对象的过程中，也就引用了此对象。

因此，对象的创建与对它们的引用是独立的。为了更清楚地看到这一点，考虑下面的代码：

```
Panda myPanda, thatPanda;
myPanda = new Panda();
thatPanda = myPanda;
```

myPanda 对象被复制到 thatPanda 变量上。由于 myPanda 和 thatPanda 只是对对象的引用，上面最后一行所做的不过就是把 myPanda 的引用赋值给 thatPanda，使得 myPanda 和 thatPanda 同时指向惟一的一个 Panda 对象。

为了与上面的操作做一个比较，考虑下面的例子：



```
Panda myPanda, thatPanda;
myPanda = new Panda();
myPanda = new Panda();
```

在上面创建了两个 Panda 类的对象。当第一个对象被创建出来的时候，这个对象立即就被引用了。但是当第二个对象被创建出来的时候，也立即就被引用了，而同时对第一个对象的引用就不存在了。显然，在以后的代码中，第一个对象也不可能再被引用了。Java 虚拟机的垃圾收集器会在某个时刻把它收集走。

20.3　Java 对象的复制

java.lang.Object.clone()方法

Java 的所有类都是从 java.lang.Object 类继承而来的，而 Object 类提供下面的方法对对象进行复制：

```
protected Object clone()
```

子类当然也可以把这个方法置换掉，提供满足自己需要的复制方法。对象的复制有一个基本问题，就是对象通常都有对其他的对象的引用。当使用 Object 类的 clone()方法来复制一个对象时，此对象对其他对象的引用也同时会被复制一份。

Java 语言提供的 Cloneable 接口只起一个作用，就是在运行时期通知 Java 虚拟机可以安全地在这个类上使用 clone()方法。通过调用这个 clone()方法可以得到一个对象的复制。由于 Object 类本身并不实现 Cloneable 接口，因此如果所考虑的类没有实现 Cloneable 接口时，调用 clone()方法会抛出 CloneNotSupportedException 异常。

系统可以在 PandaToClone 里面置换掉 java.lang.Object 提供的 clone()方法，以便控制复制的过程。在下面的例子里，PandaToClone 是一个带有几个性质的类，它将 java.lang.Object 类的克隆方法置换掉了，由一个新的 clone()方法提供复制自己实例的任务，如代码清单 1 所示。

代码清单 1：PandaToClone 类的源代码

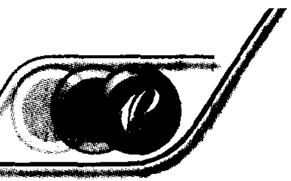
```
class PandaToClone implements Cloneable
{
    private int height, weight, age;
    /**
     * 构造子
     */
    public PandaToClone (int height, int weight)
    {
        this.age = 0;
        this.weight = weight;
        this.height = height;
```




```
    }
    /**
     * 年龄的赋值方法
     */
    public void setAge(int age)
    {
        this.age = age;
    }
    /**
     * 年龄的取值方法
     */
    public int getAge()
    {
        return age;
    }
    /**
     * 身高的取值方法
     */
    public int getHeight()
    {
        return height;
    }
    /**
     * 体重的取值方法
     */
    public int getWeight()
    {
        return weight;
    }
    /**
     * 克隆方法
     */
    public Object clone()
    {
       // 创建一个本类的对象,
       // 并返还给调用者
       PandaToClone temp =
           new PandaToClone(height, weight);

       temp.setAge(age);
       // 注意返还值的类型必须是 Object
       return (Object) temp;
    }
}
```

客户端的源代码如代码清单 2 所示。



代码清单 2：客户端的源代码

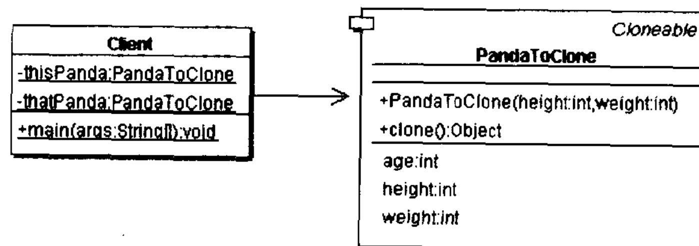
```
public class Client
{
    private static PandaToClone thisPanda , thatPanda;
    public static void main(String[] args)
    {
        thisPanda = new PandaToClone(15, 25);
        thisPanda.setAge(3);
        // Create the second object by cloning the first
        thatPanda = (PandaToClone) thisPanda.clone();
        // Now describe these objects on the system console :
        System.out.println(" Age of this panda : "
            + thisPanda.getAge());
        System.out.println("                height : "
            + thisPanda.getHeight());
        System.out.println("                weight : "
            + thisPanda.getWeight());
        System.out.println(" Age of that panda : "
            + thatPanda.getAge());
        System.out.println("                height : "
            + thatPanda.getHeight());
        System.out.println("                weight : "
            + thatPanda.getWeight());
    }
}
```

在运行时，客户端首先创建了一个 PandaToClone 的实例，并且给各个性质赋值。然后将此对象复制一份。系统运行的结果如代码清单 3 所示。

代码清单 3：运行的结果

```
Age of this panda : 3
           height : 15
           weight : 25
Age of that panda : 3
           height : 15
           weight : 25
```

从运行的结果可以看出，克隆对象与原对象的性质是完全一样的。现在考察下面这个例子，克隆熊猫的模拟系统结构图如下图所示。



可以看出，Client 对象持有一个对 PandaToClone 对象的引用，而后 Client 对象调用 PandaToClone.clone()方法，得到一份复制后的对象，这其实就是原始模型模式。

克隆满足的条件

clone()方法将对象复制了一份并返还给调用者。所谓“复制”的含义与 clone()方法是怎么实现的有关。一般而言，clone()方法满足以下的描述[BLOCH01]：

（1）对任何的对象 *x*，都有：x.clone() != x。换言之，克隆对象与原对象不是同一个对象。

（2）对任何的对象 *x*，都有：x.clone().getClass == x.getClass()，换言之，克隆对象与原对象的类型一样。

（3）如果对象 *x* 的 equals()方法定义恰当的话，那么 x.clone().equals(x)应当是成立的。

在 Java 语言的 API 中，凡是提供了 clone()方法的类，都满足上面的这些条件。Java 语言的设计师在设计自己的 clone()方法时，也应当遵守这三个条件。

一般来说，上面的三个条件中的前两个是必需的，而第三个是可选的。本书则推荐读者遵守所有的这三个条件。

equals()方法的讨论

上面所给出的三个条件中，第三个条件常常被忽视。很多读者误以为可以通过继承得到 java.lang.Object 对象的 equals()方法就足够了，而这是不对的。为了帮助读者了解 equals()方法，首先看一看 java.lang.Object 的源代码，如代码清单 4 所示。

代码清单 4：java.lang.Object 源代码片断

```
public boolean equals(Object obj)
{
    return (this == obj);
}
```

也就是说，当两个变量指向同一个对象时，equals()方法才会返还 true。很显然，这并不适合于所有需要被克隆的对象。

假设被克隆的对象按照它们的内部状态是否可变，划分成可变对象和不变对象的话，那么可变对象和不变对象所提供的 equals()方法的工作方式应当是不同的。

可变对象只有当它们是同一个对象时，equals()才会返还 true，所以这样的类型可以直接从 java.lang.Object 继承这个方法。不变对象必须含有相同的状态才可能满足这个条件，因此不变类型必须自行实现这个 equals()方法。

一个典型的例子就是 Java 的 String 对象。读过本书的“不变（Immutable）模式”一章的读者都应该知道，Java 的 String 对象都是不变对象，而不变对象具有不会改变的内部状态：具有相同内部状态的 String 对象对于客户端就没有区别。这样一来，两个 String 对象的比较就应当是它们的内部状态值的比较。

本章在后面不会接触到不变对象的克隆，因此不会提供特殊的 equals()方法。关于不



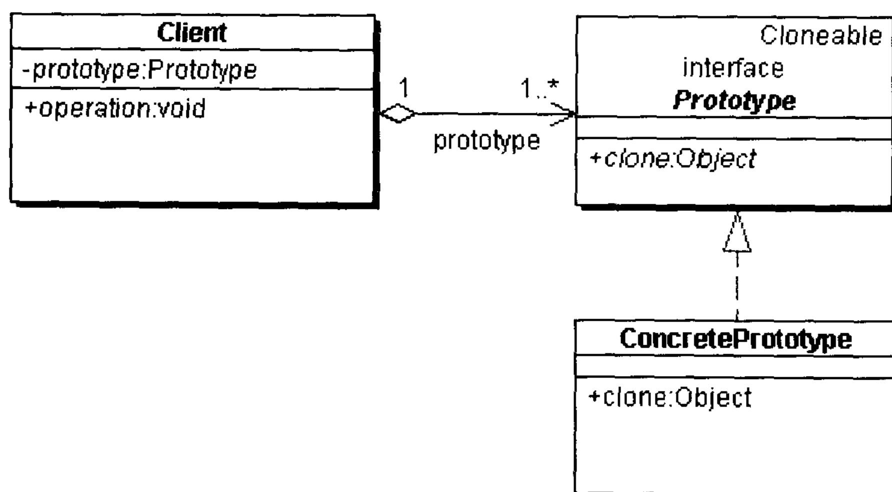
变对象和不变模式的讨论，请读者参阅本书的“不变（Immutable）模式”一章。

20.4　原始模型模式的结构

原始模型模式有两种表现形式：第一种是简单形式，第二种是登记形式。这两种表现形式仅仅是原始模型模式的不同实现，但是由于它们的区别影响了模式结构的细节，因此本书把它们提前到这里讲解。

简单形式的原始模型模式

第一种形式的原始模型模式的类图如下图所示。



这种形式涉及到三个角色：

- 客户（Client）角色：客户类提出创建对象的请求。
- 抽象原型（Prototype）角色：这是一个抽象角色，通常由一个 Java 接口或 Java 抽象类实现。此角色给出所有的具体原型类所需的接口。
- 具体原型（Concrete Prototype）角色：被复制的对象。此角色需要实现抽象的原型角色所要求的接口。

下面给出一个示意性的实现，首先是客户端的源代码，如代码清单 5 所示。

代码清单 5：客户端的源代码

```
public class Client
{
    private Prototype prototype;

    public void operation(Prototype example)
    {
        Prototype p = (Prototype) example.clone();
    }
```

```
}
```

抽象原型角色声明了一个 clone()方法，如代码清单 6 所示。

代码清单 6：抽象原型角色的源代码

```
public interface Prototype
    extends Cloneable
{
    Prototype clone();
}
```

具体原型角色实现了 clone()方法，如代码清单 7 所示。

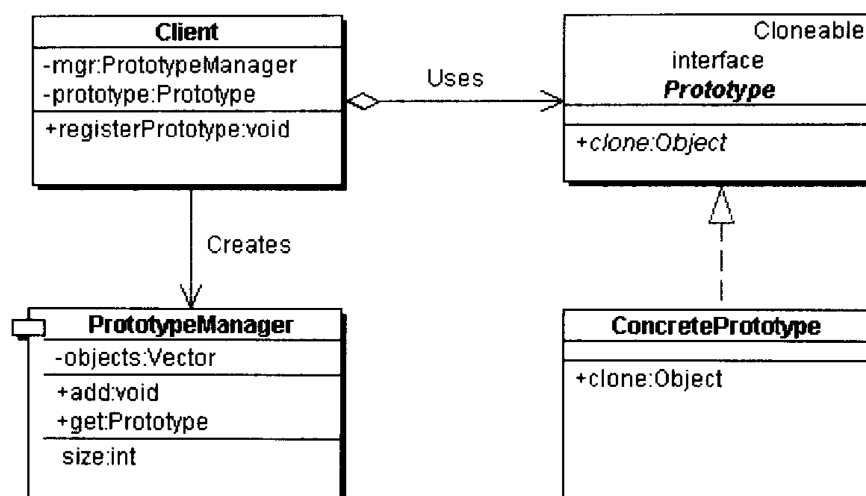
代码清单 7：具体原型角色的源代码

```
public class ConcretePrototype
    implements Prototype
{
  /**
    * 克隆方法
    */
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch(CloneNotSupportedException e)
        {
            return null;
        }
    }
}
```

在熊猫克隆的例子里，原型角色由接口 Cloneable 扮演，具体原型角色由 PandaToClone 类扮演。这意味着，通常在 Java 语言中实现 Clone 方法的办法就是这种形式的原始模型模式的实现。

登记形式的原始模型模式

第二种形式的原始模型模式的类图如下图所示。



作为原始模型模式的第二种形式，它有如下的角色：

- 客户端（Client）角色：客户端类向管理员提出创建对象的请求。
- 抽象原型（Prototype）角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体原型类所需的接口。
- 具体原型（Concrete Prototype）角色：被复制的对象。需要实现抽象的原型角色所要求的接口。
- 原型管理器（Prototype Manager）角色：创建具体原型类的对象，并记录每一个被创建的对象。

下面给出一个示意性实现的源代码。首先，抽象原型角色声明了一个方法，即 clone() 方法，如代码清单 8 所示。

代码清单 8：抽象原型角色 Prototype 的源代码

```
public interface Prototype
    extends Cloneable
{
    public Object clone();
}
```

具体原型角色实现了抽象原型角色的接口，也就是 clone()方法，如代码清单 9 所示。

代码清单 9：具体原型角色的源代码

```
public class ConcretePrototype
    implements Prototype
{
  /**
   * 克隆方法
   */
   public synchronized Object clone()
   {
       Prototype temp = null;
       try
```

```
        {
            temp = (Prototype) super.clone();
            return temp;
        }
        catch(CloneNotSupportedException e)
        {
            System.out.println("Clone failed.");
        }
        finally
        {
            return temp;
        }
    }
}
```

原型管理器角色保持一个聚集，作为对所有原型对象的登记，这个角色提供必要的方法，供外界增加新的原型对象和取得已经登记过的原型对象。原型管理器类的示意性源代码如代码清单 10 所示。

代码清单 10：原型管理器类的源代码

```
import java.util.Vector;
public class PrototypeManager
{
    private Vector objects = new Vector();

   /**
    * 聚集管理方法：增加一个新的对象
    */
    public void add(Prototype object)
    {
        objects.add(object);
    }
   /**
    * 聚集管理方法：取出聚集中的一个对象
    */
    public Prototype get(int i)
    Mb
        return (Prototype) objects.get(i);
    }
   /**
    * 聚集管理方法：给出聚集的大小
    */
    public int getSize()
    {
        return objects.size();
    }
```

```
}
```

客户端角色 Client 类的源代码如代码清单 11 所示。

代码清单 11：Client 类的源代码

```
public class Client
{
    private PrototypeManager mgr;
    private Prototype prototype;
    public void registerPrototype()
    {
        prototype = new ConcretePrototype();
        Prototype copytype = (Prototype)
            prototype.clone();
        mgr.add(copytype);
    }
}
```

可以看出，上面的这个客户对象首先创建了一个新的原型对象，然后将之克隆一份，并且存到原型管理器中。

两种形式的比较

简单形式和登记形式的原始模型模式各有其长处和短处。

如果需要创建的原型对象数目较少而且比较固定的话，可以采取第一种形式，也即简单形式的原始模型模式。在这种情况下，原型对象的引用可以由客户端自己保存。

如果要创建的原型对象数目不固定的话，可以采取第二种形式，也即登记形式的原始模型模式。在这种情况下，客户端并不保存对原型对象的引用，这个任务被交给管理员对象。在复制一个原型对象之前，客户端可以查看管理员对象是否已经有一个满足要求的原型对象。如果有，可以直接从管理员类取得这个对象引用；如果没有，客户端就需要自行复制此原型对象。

20.5　模式的实现：深复制和浅复制

正如前面所说的，复制或克隆有两种方式。这两种方式分别叫做浅复制（浅克隆）和深复制（深克隆）。

浅复制（浅克隆）

被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用都仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

深复制（深克隆）

被复制对象的所有的变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深复制把要复制的对象所引用的对象都复制了一遍，而这种对被引用到的对象的复制叫做间接复制。

深复制要深入到多少层，是一个不易确定的问题。在决定以深复制的方式复制一个对象的时候，必须决定对间接复制的对象是采取浅复制还是继续采用深复制。因此，在采取深复制时，需要决定多深才算深。此外，在深复制的过程中，很可能会出现循环引用的问题，必须小心处理。

利用串行化来做深复制

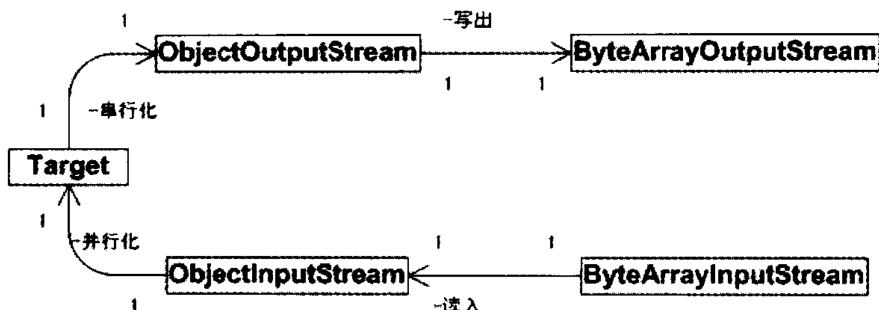
把对象写到流里的过程是串行化（Serilization）过程，但是在 Java 程序师圈子里又非常形象地称为“冷冻”或者“腌咸菜（pickling）”过程；而把对象从流中读出来的并行化（Deserialization）过程则叫做“解冻”或者“回鲜”（depickling）过程。应当指出的是，写到流里的是对象的一个拷贝，而原对象仍然存在于 JVM 里面，因此“腌成咸菜”的只是对象的一个拷贝，Java 咸菜还可以回鲜。

在 Java 语言里深复制一个对象，常常可以先使对象实现 Serializable 接口，然后把对象（实际上只是对象的一个拷贝）写到一个流里（腌成咸菜），再从流里读回来（把咸菜回鲜），便可以重建对象，如代码清单 12 所示。

代码清单 12：深复制的源代码

```
public Object deepClone()
{
    //将对象写到流里
    ByteArrayOutputStream bo =
         new ByteArrayOutputStream();
    ObjectOutputStream oo =
         new ObjectOutputStream(bo);
    oo.writeObject(this);
    //从流里读回来
    ByteArrayInputStream bi =
         new ByteArrayInputStream(bo.toByteArray());
    ObjectInputStream oi =
         new ObjectInputStream(bi);
    return (oi.readObject());
}
```

这个串行化过程可以用下图所示的结构图表述。



这样做的前提就是对象以及对象内部所有引用到的对象都是可串行化的，否则，就需要仔细考察那些不可串行化的对象可否设成 transient，从而将之排除在复制过程之外。

浅复制显然比深复制更容易实现，因为 Java 语言的所有类都会继承一个 clone()方法，而这个 clone()方法所做的正是浅复制。

有一些对象，比如线程（thread）对象或 Socket 对象，是不能简单复制或共享的。不管是使用深复制还是浅复制，只要涉及这样的间接对象，就必须把间接对象设成 transient 而不予复制；或者由程序自行创建出相当的同种对象，权且当做复制件使用。

模式的实现：选择哪一种形式

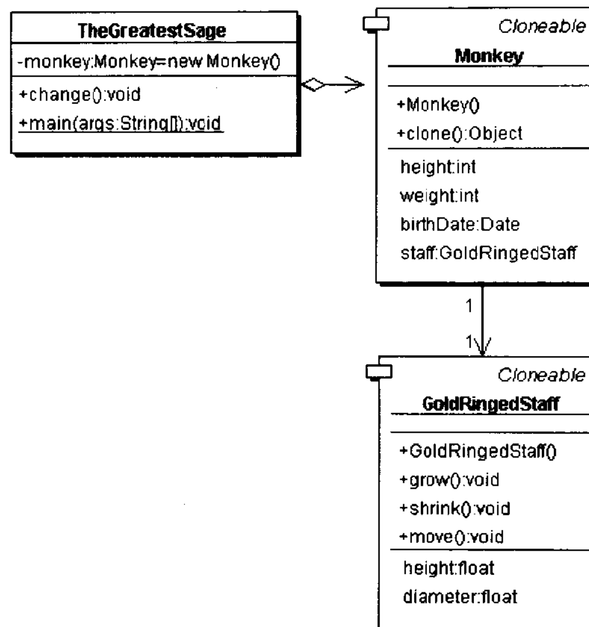
如果客户角色所使用的原型对象只有固定的几个，那么使用单独的变量来引用每一个原型对象就很方便，这时候简单形式的原始模型模式就比较合适。而如果客户端所创建的原型对象的个数不是固定的，而是动态地变化的，那么就不妨使用一个 Vector 类型的变量，用来动态地存储和释放原型对象。这时候，使用第二种形式的原始模型模式就比较合乎需要。

20.6　孙大圣的身外身法术

浅复制

孙大圣的身外身本领如果在 Java 语言里使用原始模型模式来实现的话，会怎样呢？首先，齐天大圣（The Greatest Sage）即 TheGreatestSage 类扮演客户角色。齐天大圣持有一个猢狲（Monkey）的实例，而猢狲就是大圣本尊。Monkey 类具有继承自 java.lang.Object 的 clone()方法，因此，可以通过调用这个克隆方法来复制一个 Monkey 实例。

使用浅复制实现孙大圣身外身法术的设计图如下图所示。



孙大圣本人由 TheGreatestSage 类代表，如代码清单 13 所示。

代码清单 13：孙大圣类的源代码

```
public class TheGreatestSage
{
    private Monkey monkey = new Monkey();

    public void change()
    {
        // 创建大圣本尊对象
        Monkey copyMonkey;
        // 空循环一会儿
        for (int i = 0 ; i < 2000; i++){ }
        // 克隆大圣本尊
        copyMonkey = (Monkey) monkey.clone();
        System.out.println("Monkey King's birth date="
            + monkey.getBirthDate() );
        System.out.println("Copy monkey's birth date="
            + copyMonkey.getBirthDate() );
        System.out.println("Monkey King == Copy Monkey? "
            + (monkey == copyMonkey));
        System.out.println("Monkey King's Staff == Copy Monkey's Staff? " +
            (monkey.getStaff() == copyMonkey.getStaff()));
    }
    public static void main(String[] args)
    {
        TheGreatestSage sage =
            new TheGreatestSage();
        sage.change();
```



```
    }
}
```

大圣本尊由 Monkey 类代表，这个类扮演具体原型角色，如代码清单 14 所示。

代码清单 14：大圣本尊、猢狲类的源代码

```
import java.util.Date;
public class Monkey implements Cloneable
{
    private int height;
    private int weight;
    private GoldRingedStaff staff;
    private Date birthDate;

   /**
    * 构造子
    */
    public Monkey()
    {
        this.birthDate = new Date();
    }
   /**
    * 克隆方法
    */
    public Object clone()
    {
        Monkey temp = null;
        try
        {
            temp = (Monkey) super.clone();
        }
        catch(CloneNotSupportedException e)
        {
            System.out.println("Clone failed");
        }
        finally
        {
            return temp;
        }
    }
   /**
    * 身高的取值方法
    */
    public int getHeight()
    {
        return height;
    }
```

```
    /**
     * 身高的赋值方法
     */
     public void setHeight(int height)
     {
         this.height = height;
     }
    /**
     * 体重的取值方法
     */
     public int getWeight()
     {
         return weight;
     }
    /**
     * 体重的赋值方法
     */
     public void setWeight(int weight)
     {
         this.weight = weight;
     }
    /**
     * 生日的取值方法
     */
     public Date getBirthDate()
     {
         return birthDate;
     }
    /**
     * 生日的赋值方法
     */
     public void setBirthDate(Date birthDate)
     {
         this.birthDate = birthDate;
     }
}
```

可以看到，大圣本尊持有一个金箍（Gold Ringed）棒（Staff）的实例。金箍棒类 GoldRingedStaff 的源代码如代码清单 15 所示。

代码清单 15：金箍棒类的源代码

```
public class GoldRingedStaff
{
    private float height = 100.0F;
    private float diameter = 10.0F;

   /**
```

```
     * 构造子
     */
    public GoldRingedStaff()
    {
        // write your code here
    }

    /**
     * 增长行为，每次调用长度和半径增加一倍
     */
    public void grow()
    {
        this.diameter *= 2.0;
        this.height *= 2;
    }
    /**
     * 缩小行为，每次调用长度和半径减少一半
     */
    public void shrink()
    {
        this.diameter /= 2;
        this.height /= 2;
    }
    /**
     * 移动
     */
    public void move()
    {
        //write your code for moving the staff
    }
    /**
     * 高度的取值方法
     */
    public float getHeight()
    {
        return height;
    }
    /**
     * 高度的赋值方法
     */
    public void setHeight(float height)
    {
        this.height = height;
    }
    /**
     * 半径的取值方法
```

```
    */
    public float getDiameter()
    {
        return diameter;
    }
   /**
    * 半径的赋值方法
    */
    public void setDiameter(float diameter)
    {
        this.diameter = diameter;
    }
}
```

可以看到，金箍棒具有长度（height）、直径（diameter）两个性质，以及伸缩（shrink）、移动（move）两个方法。

当运行 TheGreatestSage 类时，main()方法会首先创建大圣本尊对象，然后在空循环 2 000 次以后，浅复制大圣本尊对象。程序在运行时打印出的信息如代码清单 16 所示。

代码清单 16：运行的结果

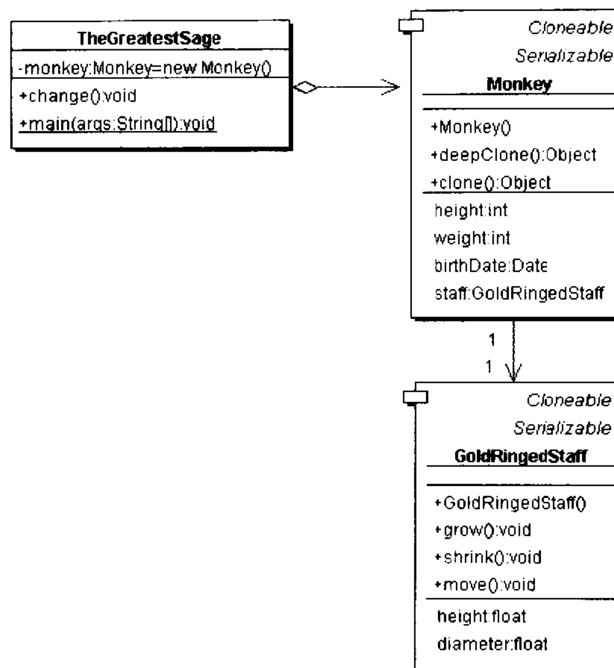
```
Monkey King's birth date=Wed Jan 16 21:58:26 EST 2002
Copy monkey's birth date=Wed Jan 16 21:58:26 EST 2002
Monkey King == Copy Monkey? false
Monkey King's Staff == Copy Monkey's Staff? true
```

可以看出，首先，复制的大圣本尊具有和原始的大圣本尊对象一样的 birthDate，而本尊对象并不相等，这表明他们二者是克隆的关系；其次，复制的大圣本尊所持有的金箍棒和原始的大圣本尊所持有的金箍棒是一样的，这表明二者所持有的金箍棒根本是一根，而不是两根。

正如前面所述，继承自 java.lang.Object 类的 clone()方法是浅复制。换言之，齐天大圣的所有化身所持有的金箍棒引用全都是指向一个对象的。这样齐天大圣的金箍棒要被所有的猢狲耍，与《西游记》中的描写并不一致。要纠正这一点，就需要考虑使用深复制。

深复制

为做到深复制，所有需要复制的对象都需要实现 java.io.Serializable 接口。系统的设计如下图所示。



孙大圣的源代码如代码清单 17 所示。

代码清单 17：孙大圣类的源代码

```
import java.util.Date;
import java.io.IOException;
import java.lang.ClassNotFoundException;
public class TheGreatestSage
{
    private Monkey monkey = new Monkey();

    public void change() throws IOException,
        ClassNotFoundException
    {
        Monkey copyMonkey;
        for (int i = 0 ; i < 2000; i++){ }
        copyMonkey = (Monkey) monkey.deepClone();
        System.out.println("Monkey King's birth date="
            + monkey.getBirthDate() );
        System.out.println("Copy monkey's birth date="
            + copyMonkey.getBirthDate() );
        System.out.println("Monkey King == Copy Monkey? "
            + (monkey == copyMonkey));
        System.out.println("Monkey King's Staff == Copy Monkey's Staff? " +
            (monkey.getStaff() == copyMonkey.getStaff()));
    }
    public static void main(String[] args)
        throws IOException, ClassNotFoundException
    {
```

```
        TheGreatestSage sage = new TheGreatestSage();
        sage.change();
    }
}
```

在大圣本尊 Monkey 类里面，有两个克隆方法，一个是 clone()，也即浅克隆；另一个是 deepClone()，也即深克隆。在深克隆方法里，大圣本尊对象（一个拷贝）被腌成咸菜，然后又被回鲜。回鲜的对象就成了一个深克隆的结果，如代码清单 18 所示。

代码清单 18：大圣本尊、猢狲类的源代码

```
import java.util.Date;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.OptionalDataException;
import java.io.Serializable;
public class Monkey
    implements Cloneable, Serializable
{
    private int height;
    private int weight;
    private GoldRingedStaff staff;
    private Date birthDate;

   /**
    * 构造子
    */
    public Monkey()
    {
        this.birthDate = new Date();
        this.staff = new GoldRingedStaff();
    }
   /**
    * 深克隆方法
    */
    public Object deepClone()
        throws IOException, OptionalDataException,
          ClassNotFoundException
    {
        //首先将对象写出到流里
        ByteArrayOutputStream bo =
              new ByteArrayOutputStream();
        ObjectOutputStream oo =
              new ObjectOutputStream(bo);
```

```
        oo.writeObject(this);
        //然后将对象从流里读出来
        ByteArrayInputStream bi =
            new ByteArrayInputStream(bo.toByteArray());
        ObjectInputStream oi =
            new ObjectInputStream(bi);
        return (oi.readObject());
    }
    /**
     * 浅克隆方法
     */
    public Object clone()
    {
        Monkey temp = null;
        try
        {
            temp = (Monkey) super.clone();
        }
        catch(CloneNotSupportedException e)
        {
            System.out.println("Clone failed");
        }
        finally
        {
            return temp;
        }
    }
    /**
     * 身高的取值方法
     */
    public int getHeight()
    {
        return height;
    }
    /**
     * 身高的赋值方法
     */
    public void setHeight(int height)
    {
        this.height = height;
    }
    /**
     * 体重的取值方法
     */
    public int getWeight()
    {
```

```
        return weight;
    }
    /**
     * 体重的赋值方法
     */
    public void setWeight(int weight)
    {
        this.weight = weight;
    }
    /**
     * 生日的取值方法
     */
    public Date getBirthDate()
    {
        return birthDate;
    }
    /**
     * 生日的赋值方法
     */
    public void setBirthDate(Date birthDate)
    {
        this.birthDate = birthDate;
    }
    /**
     * 金箍棒的取值方法
     */
    public GoldRingedStaff getStaff()
    {
        return staff;
    }
}
```

请注意新的方法 deepClone()和此类所实现的 Serializable 接口。

可以看到，大圣本尊持有一个金箍棒（GoldRingedStaff）的实例。在大圣复制件里面，此金箍棒实例是原大圣本尊对象所持有的金箍棒对象的一个拷贝。在大圣本尊对象被“冷藏”和“解冻”时，它所持有的金箍棒对象也同时被“冷藏”和“解冻”，这使得复制的大圣的金箍棒和原大圣本尊对象所持有的金箍棒对象是两个独立的对象。金箍棒的源代码如代码清单 19 所示。

代码清单 19：金箍棒类的源代码

```
import java.util.Date;
import java.io.Serializable;
public class GoldRingedStaff
    implements Cloneable, Serializable
{
    private float height = 100.0F;
```



```
    private float diameter = 10.0F;

   /**
    * 构造子
    */
    public GoldRingedStaff()
    {
        //write your code here
    }
   /**
    * 增长行为，每次调用长度和半径增加一倍
    */
    public void grow()
    {
        this.diameter *= 2.0;
        this.height *= 2;
    }
   /**
    * 缩小行为，每次调用长度和半径减少一半
    */
    public void shrink()
    {
        this.diameter /= 2;
        this.height /= 2;
    }
   /**
    * 移动
    */
    public void move()
    {
        //write your code for moving the staff
    }
   /**
    * 高度的取值方法
    */
    public float getHeight()
    {
        return height;
    }
   /**
    * 高度的赋值方法
    */
    public void setHeight(float height)
    {
        this.height = height;
    }
```

```
    /**
     * 半径的取值方法
     */
    public float getDiameter()
    {
        return diameter;
    }
    /**
     * 半径的赋值方法
     */
    public void setDiameter(float diameter)
    {
        this.diameter = diameter;
    }
}
```

注意 它所实现的 Serializable 接口表明此类可以被安全地复制。

当运行 TheGreatestSage 类时，首先创建大圣本尊对象，然后在空循环 2 000 次以后，对大圣本尊对象进行深复制。程序输出的信息如代码清单 20 所示。

代码清单 20：运行的结果

```
Monkey King's birth date=Wed Jan 16 22:15:32 EST 2002
Copy monkey's birth date=Wed Jan 16 22:15:32 EST 2002
Monkey King == Copy Monkey? false
Monkey King's Staff == Copy Monkey's Staff? false
```

从运行的结果可以看出，大圣的金箍棒和他的身外之身的金箍棒是不同的对象。这是因为使用了深复制，从而把大圣本尊所引用的对象也都复制了一遍，其中也包括金箍棒。

20.7 在什么情况下使用原始模型模式

假设一个系统的产品类是动态加载的，而且产品类具有一定的等级结构。这个时候如果采取工厂模式的话，工厂类就不得不具有一个相应的等级结构。而产品类的等级结构一旦变化，工厂类的等级结构就不得不有一个相应的变化。这对于产品结构可能会有经常性变化的系统来说，采用工厂模式就有不方便之处。

这时如果采取原始模型模式，给每一个产品类配备一个克隆方法（大多数的时候只需给产品类等级结构的根类配备一个克隆方法），便可以避免使用工厂模式所带来的具有固定等级结构的工厂类。

这样，一个使用了原始模型模式的系统与它的产品对象是怎么创建出来的，以及这些产品对象之间的结构是怎样的，以及这个结构会不会发生变化是没有关系的。



20.8　原始模型模式的优点和缺点

抽象工厂模式有许多与原始模型模式和建造模式相同的效果，包括客户端不知道具体产品类，而只知道抽象产品类，客户端不需要知道这么多的具体产品名称。如果有新的产品类加入，客户端不需要进行改造就可直接使用。

原始模型模式有其特有的优点：

（1）原始模型模式允许动态地增加或减少产品类。由于创建产品类实例的方法是产品类内部具有的，因此，增加新产品对整个结构没有影响。

（2）原始模型模式提供简化的创建结构。工厂方法模式常常需要有一个与产品类等级结构相同的等级结构，而原始模型模式就不需要这样。对于 Java 设计师来说，原始模型模式又有其特有的方便之处，因为 Java 语言天生就将原始模型模式设计到了语言模型里面。善于利用原始模型模式和 Java 语言的特点，可以事半而功倍。

（3）具有给一个应用软件动态加载新功能的能力。例如，一个分析 Web 服务器的记录文件的应用软件，针对每一种记录文件格式，都可以由一个相应的“格式类”负责。如果出现了应用软件所不支持的新的 Web 服务器，只需要提供一个格式类的克隆，并在客户端登记即可，而不必给每个软件的用户提供一个全新的软件包。

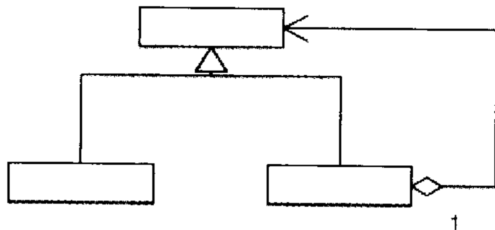
（4）产品类不需要非得有任何事先确定的等级结构，因为原始模型模式适用于任何的等级结构。

原始模型模式最主要的缺点是每一个类都必须配备一个克隆方法。配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类来说不是很难，而对于已经有的类不一定很容易，特别是当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。

20.9　原始模型模式与其他模式的关系

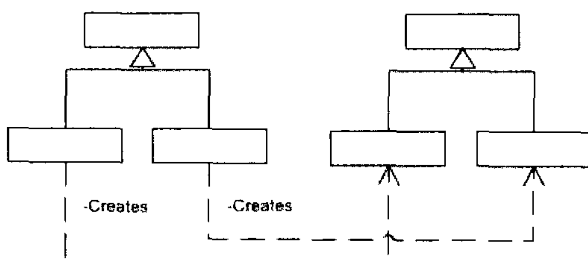
原始模型模式与合成（Composite）模式的关系

原始模型模式经常与合成模式一同使用，因为原型对象经常是合成对象。合成模式的简略类图如下图所示。



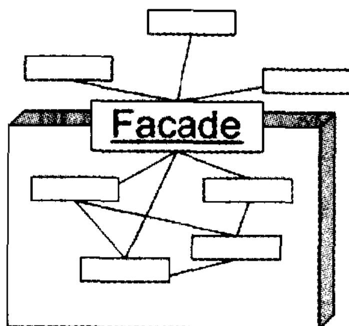
原始模型模式与抽象工厂（Abstract Factory）模式的关系

如果系统不需要动态地改变原型对象，抽象工厂模式可以成为原始模型模式的替代品。抽象工厂模式的简略类图如下图所示。



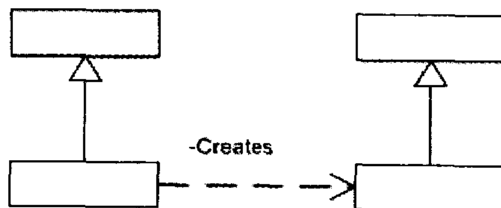
原始模型模式与门面（Facade）模式的关系

原始模型模式的客户端通常可以将系统的其他对象与参与原始模型模式的对象分隔开，起到一个门面对象的作用。门面模式的简略类图如下图所示。



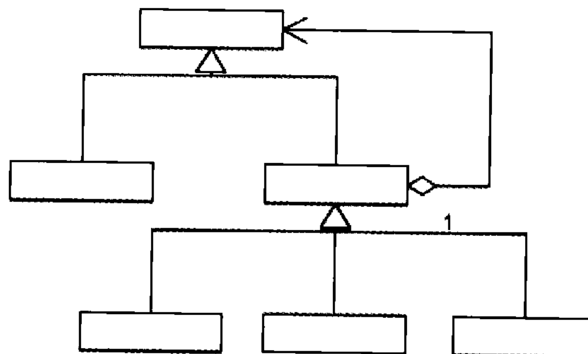
原始模型模式与工厂方法（Factory Method）模式的关系

如果原型对象只有一种，而且不会增加的话，工厂方法模式可以成为一种替代模式。工厂方法模式的简略类图如下图所示。

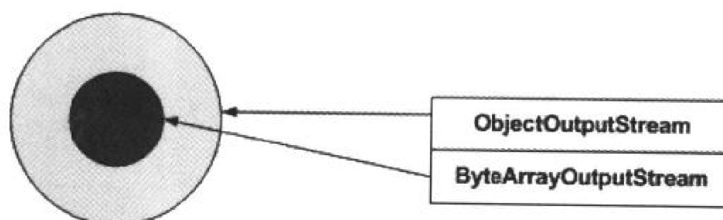


原始模型模式与装饰（Decorator）模式的关系

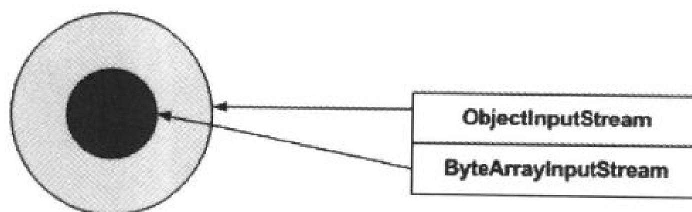
原始模型模式常常与装饰模式一同使用。装饰模式的简略类图如下图所示。



Java 的 IO 库广泛使用了装饰模式，串行化的过程也是一个使用装饰模式的过程，输出端的包裹过程可以使用下面的输出端和输入端的包裹示意图描述。输出端的包裹过程如下图所示。



输入端的包裹过程如下图所示。



关于装饰模式的详细讨论请见本书的“装饰（Decorator）模式”一章。

参考文献

[BLOCH01] Joshua Bloch. Effective Java–Programming Language Guide. published by Addison-Wesley，2001

第21章　专题：JavaBean的“冷藏”和“解冻”

本章主要讲解怎样使用java.beans.Beans类对经过串行化处理的JavaBean进行“解冻”处理。在本章读者会遇到工厂方法模式、标识接口模式和原始模型模式。

希望读者在阅读本章之前，首先熟悉一下Java语言的串行化功能，并阅读本书的“Java接口”、“工厂方法（Factory Method）模式”以及“原始模型（Prototype）模式”等章节。

21.1　什么是“冷藏”和“解冻”

Java的串行化功能

在Java语言从1.0升级到1.1版本时，Java语言的功能在很多方面得到了很大的提高，串行化（Serialization）就是那时候引进的新的功能。用一句话来讲，串行化使得一个程序可以把一个完整的对象写到一个Byte流里面，或者从一个Byte流里面读出一个事先存储在里面的完整的对象；串行化可以把Java对象和原始数据类型转换成一个适合于某种网络或文件系统的Byte流。

串行化处理的威力

人们对串行处理功能往往有两种极端化的看法：一些人认为它是很强大和重要的功能；另外有些人则相反，认为串行化是比较不重要的功能。

串行处理功能真正强大之处在于一个Java程序不需要直接处理存储在硬盘上面的原始数据，就可以很容易地将一个Java对象和一个二进制流之间相互转换。任何一个接触过Pascal或者C语言的程序员都知道从硬盘上面读取数据的麻烦，更不用说将二进制流转换成为所需要格式的各种细节的繁琐。现在，Java语言将这些麻烦和繁琐的细节统统省去了。

那么，使用最笼统的语言来描述，Java语言的串行化意味着什么呢？将一个Java对象串行化只需要实现两个接口中的一个即可：要么实现java.io.Serializable接口，要么实现Externalizable接口。Serializable接口是一个标识接口，也就是说它是一个没有方法的接口。Externalizable接口是一个有两个方法的Serializable接口的子接口，因此实现这个接口的类必须实现这两个方法。由于实现Serializable接口更为普遍，因此，本章主要讨

论 Serializable 接口的实现。

关于怎样将一个 JavaBean 串行化的内容，可以参见本书的“原始模型（Prototype）模式”一章。

Serializable 接口是标识接口

读者如果想进一步了解标识接口的结构和使用方法，请见本书的“Java 接口”一章。

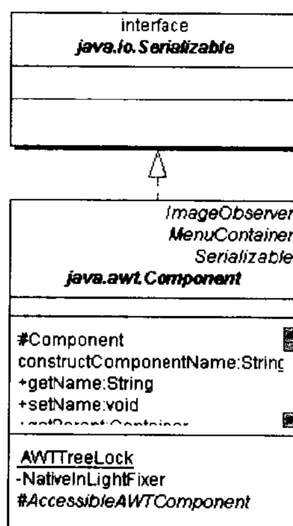
使用较为严格的语言来描述，任何 Java 对象的串行化的最低要求是它必须实现 Serializable 接口。这个接口并没有声明任何方法，实现这个接口只意味着设计师已经考虑了串行化会给这个类带来的效果，并指示 Java 虚拟机这个类是可以串行化的。如果一个对象的内部状态不适合输出到硬盘上，或者不适合通过网络传送，那么设计师就应当慎重处理这个类的设计，具体情况请见下面的讨论。对一个对象实施串行化处理会导致这个对象所引用到的对象和原始类型同时受到串行化处理。那些因为被引用而被串行化的对象的内部状态也同样会被保存起来。当并行化（deserialize）发生时，被串行化的对象的内部状态就会被恢复。

正如本书在前面所提到的，串行化（Serialization）又叫做“腌制”或者“冷藏”；并行化（Deserialization）又叫做“回鲜”或者“解冻”。

21.2 什么可以“冷藏”

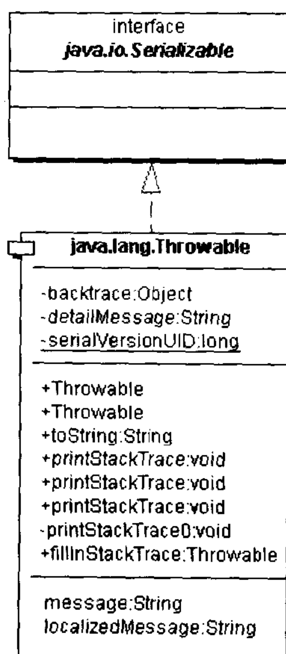
什么类可以串行化

java.awt.Component 实现了 Serializable 接口，如下图所示。



因此，所有 Component 直接的和间接的子类，包括 Button、Scrollbar、TextArea、List、

Container、Panel、java.applet.Applet 以及所有的 Applet 的子类和 Swing 的子类，全都是可以串行化的。再比如，java.lang.Throwable 类实现了 Serializable 接口，如下图所示。



因此，所有的 Exception 和 Error 类均是可以串行化的。

一般而言，Exception、Error 以及其他继承自 Throwable 的类均是可以串行化的；而流、所有的 Reader 和 Writer 以及其他的 I/O 类均是不可以串行化的；AWT 和 Swing、容器类、事件类均是可以串行化的；事件适配器类、图像过滤器类、AWT 包中与操作系统相关的特性类均是不可以串行化的；原始类型的封装类中只有 Void 类是可以串行化的；多数的 java.lang 包中的类是不可以串行化的；反射（Reflection）类是不可以串行化的；java.math 中的类都是可以串行化的；压缩类都是不可以串行化的。

什么样的类不可以串行化

那么满足什么条件的类是不可以串行化的呢？

一般而言，满足下面的四个条件之一的类就不应当串行化：

（1）一个类与本地代码（native code）有紧密的关系。比如 java.util.zip.Deflater 就是一个例子。

（2）对象的内部状态依赖于 Java 虚拟机或运行环境，从而在每一次运行时这个状态都有可能不同。比如 java.lang.Thread，java.io.InputStream，java.io.FileDescriptor，java.awt.PrintJob 等。

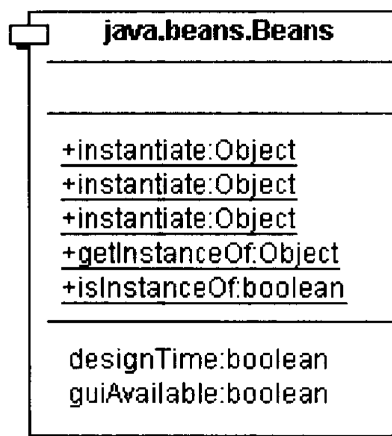
（3）串行化可能带来潜在的安全隐患。比如，java.lang.SecurityManager 以及 java.security.MessageDigest 等。

（4）一个类仅仅是一些静态方法的存放地，并没有任何的内部状态。比如，java.beans.Beans 和 java.lang.Math 便是此类。

21.3 Beans.instantiate()方法

所有 Java 的继承环境均提供 JavaBean 的可视设计环境。用户可以在这个可视设计环境中设计一个 JavaBean 的属性，比如颜色、大小等。设计的结果可以存储起来，下次可以在同样的设计环境中继续设计。读者是否想过，这样的可视设计是怎样存储起来的呢？

这就是 java.beans.Beans 类提供的功能，这个类的类图如下图所示。由图中可知，java.beans.Beans 类提供了 instantiate()方法。



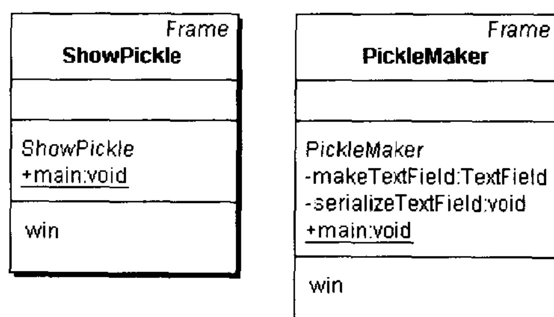
工厂方法模式

Beans.instantiate()方法是工厂方法模式的应用。通过 Beans.instantiate()方法将对象的创建过程抽象化。Beans 类在多态性原则基础上提供了几个不同特征的 instantiate()方法，这些方法都需要一个参量，即被“冷冻”的*.ser 文件的名字。如果这样的一个文件存在的话，instantiate()方法会把这个“冷冻”过的 JavaBean 重新“解冻”，即重新实例化；如果这样一个文件并不存在，而这个参量名是一个类名的话，这个类会被实例化。这时，调用这个方法的用户端并不知道一个 Bean 是已经被串行化并将要被并行化的，还是需要实例化的。

工厂方法模式将“解冻”和实例化的细节封装起来，客户端则不需要设计创建实例的细节，这就是工厂方法模式的威力。

使用方法

多数的 Java 设计师可能永远不会直接用到 java.beans.Beans 类。虽然这个类主要是为 Java 的继承环境设计的，但是普通的 Java 程序也可以利用这个类对一个 JavaBean 进行串行化。下面就给出一个例子，这个例子中有两个类，这里给出这两个类的类图，如下图所示，以说明这个方法是怎样使用的。



读者可以看到两个类，一个是 PickleMaker，另一个是 ShowPickle。在英文中，pickle 是腌渍物、泡菜的意思。通常说英文的 Java 设计师将串行化（Serialization）叫做 pickling（腌制），把并行化（Deserialization）叫做 depickling（回鲜）。

一个“冷藏”的例子

首先给出 PickleMaker 的源代码，如代码清单 1 所示。这里要做的是“冷藏”，是一个 TextField。

代码清单 1：PickleMaker 的源代码

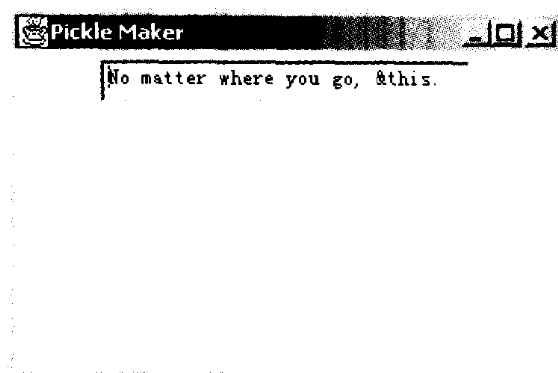
```
package com.javapatterns.serializable.instantiate;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.awt.TextField;
public class PickleMaker extends Frame
{
    /**
     * 这个构造子创建一个 frame 视窗，显示一个 TextField。
     */
    PickleMaker(String text, int size)
    {
        /*
         * （1）调用超类的构造子
         * （2）增加一个事件监听对象
         * （3）将布局管理器改为 Flow 类型
         */
        super("Pickle Maker");
        addWindowListener(new win());
        setLayout(new FlowLayout());
        /*
         * 创建 TextField 对象并将它串行化
         */
        TextField textField = makeTextField(text, size);
        serializeTextField(textField, "mytextfield.ser");
        add(textField);
```

```
}
/**
 * 这个方法创建一个 TextField 对象，以传进的 text 为文字内容，
 * 以传进的 size 为宽度
 */
private TextField makeTextField(
      String text, int size)
{
    TextField textField =
          new TextField(text, size);
    return textField;
}
/**
 * 将传进来的 TextField 对象串行化
 */
private void serializeTextField(
      TextField textField, String filename)
{
    try
    {
        FileOutputStream outStream =
              new FileOutputStream(filename);
        ObjectOutputStream out =
              new ObjectOutputStream(outStream);
        out.writeObject(textField);
        out.flush();
        out.close();
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}
/**
 * 这个方法创建一个 frame 对象，并显示一个 TextField
 */
public static void main(String[] args)
{
        Frame frame = new PickleMaker(
              "No matter where you go, &this.", 25);
        frame.setBounds(0,0,300,200);
        frame.setVisible(true);
}
/**
 * 这个内部类提供了监听功能，以便在接到操作系统传来
 * 的关闭窗口的事件时，将应用程序结束掉
```



```
 */
class win extends WindowAdapter
{
    public void windowClosing(WindowEvent evt)
    {
        Frame frame = (Frame)evt.getSource();
        frame.setVisible(false);
        frame.dispose();
        System.exit(0);
    }
}
}
```

在这个类的 main()方法被调用时，会创建一个 Frame 对象，并显示一个 TextField 对象。与此同时，这个 TextField 对象会被串行化，存储到名为 mytextfield.ser 的文件里。由于程序并没有指明一个包路径，这个文件会被放到包的根路径上。视窗上面的 TextField 对象会被冷藏，如下图所示。



一个“解冻”的例子

下面是 ShowPickle 类的源代码，如代码清单 2 所示。

代码清单 2：ShowPickle 的源代码

```
package com.javapatterns.serializable.instantiate;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
/**
 * 这个类可以被用来显示事先被串行化的 TextField 对象
 */
class ShowPickle extends Frame
{
    /**
     * 此构造子创建一个 frame 视窗，将其布局改为 FlowLayout，
     * 并将传入的 TextField 并行化，然后加到视窗上
```




```
 */
ShowPickle(String serComponent)
{

    /*
    * 调用超类的构造子，加上一个监听器对象，
    * 以便监听视窗的关闭事件。
    */
    super("Show Pickle");
    addWindowListener(new win());
    setLayout(new FlowLayout());

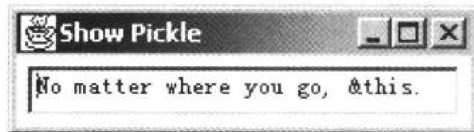
   /**
    * 将串行化的 TextField 对象实例化
    * 如果找不到串行化的对象，
    * 则显示一个普通的 TextField
    */
    TextField text;

    try
    {
      text = (TextField)Beans.instantiate(
           null, serComponent);
    }
    catch (Exception e)
    {
       System.out.println(e);
       text = new TextField();
    }
    add(text);
}
/**
 *
 * 这个方法创建一个 Frame 对象，显示并行化后的 TextField 对象
 */
public static void main(String[] args)
{
   Frame frame = new ShowPickle("mytextfield");
   frame.pack();
   frame.setVisible(true);
}
/**
* 这个内部类提供了监听功能，以便在接到操作系统传来
* 的关闭窗口的事件时，将应用程序结束掉
*/
class win extends WindowAdapter
```

```
    {
        public void windowClosing(WindowEvent evt)
        {
            Frame frame = (Frame)evt.getSource();
            frame.setVisible(false);
            frame.dispose();
            System.exit(0);
        }
    }
}
```

这个程序会显示一个 Frame 视窗，同时在视窗中显示一个 TextField 对象。这个 TextField 对象便是从“冷藏”状态下被“解冻”、重新实例化的。在这个简单的例子里，读者可以看到，java.beans.Beans.instantiate()方法接收一个（不含扩展名的）文件名，并根据这个文件名寻找这个文件。当找到这个文件后，此方法会将这个文件所包含的处于“冷藏”状态下的 JavaBean 重新实例化。

系统在运行时的情况如下图所示，TextField 对象是被重新解冻的冷藏 JavaBean。



21.4　怎样在 JSP 中使用 Beans.Instantiate()方法

在 JavaServer Pages 中调用 JavaBean 有三个标准的标识符，那就是：<jsp:useBean>、<jsp:setProperty>，以及<jsp:getProperty>。

<jsp:useBean>标识符

在<jsp:useBean>标识符里可以指明一个被“冷冻”过的.ser 文件，从而将它“解冻”。其具体语法如下：

```
<jsp:useBean id="name"
  scope="page|request|session|application"
  typeSpec>
  body
</jsp:useBean>
```

其中，typeSpec 定义如下：

```
typeSpec ::=class="className"
| class="className" type="typeName"
| type="typeName" class="className"
| beanName="beanName" type="typeName"
```

| type="typeName" beanName="beanName"
| type="typeName"

<jsp:useBean>标识符中相关属性的含义如下表所示。

属　性	含　义
Id	Id 属性是 JavaBean 对象的惟一标识，代表了一个 JavaBean 对象的实例
Scope	Scope 属性代表了 Javabean 对象的生存时间，可以是 page、request、session 和 application 中的一种
Class	代表了 JavaBean 对象的 class 名字
beanName	BeanName 属性代表了一个 Bean 串行化之后的名字，通常通过 java.beans.Beans 的 instantiate() 方法来初始化

在 JavaServer Pages 中通过 Id 来识别 JavaBean 的实例名。Id 属性具有特定的存在范围（page|request|session|application）。

BeanName 便是设计用来"解冻"一个被"冷冻"过的 JavaBean 的，虽然 java.beans.Beans 并没有明显出现在 JSP 代码里面，但是 JSP 引擎实际上是调用 java.beans.Beans 的 instantiate() 方法处理"解冻"的。

一个例子

作为一个例子，需考虑下面的 JSP 代码：

```
<jsp:useBean id="programmer" type="com.abccompany.Programmer"
    beanName="storedprogrammer" />
<jsp:getProperty name="programmer" property="salary" />
```

这段代码会把事先"冷冻"到文件"storedprogrammer.ser"中的 com.abccompany.Programmer 类的实例重新"解冻"，并调用其名为 salary 的性质。



文件名是"storedprogrammer.ser"，在 BeanName 中指明的是"storedprogrammer"，而没有扩展名。这些"冷冻"文件必须放到"WEB-INF\jsp\beans"目录下才能被找到。

这个例子里，一个 programmer 对象首先被"冷藏"起来，然后又被"解冻"，重新实例化。

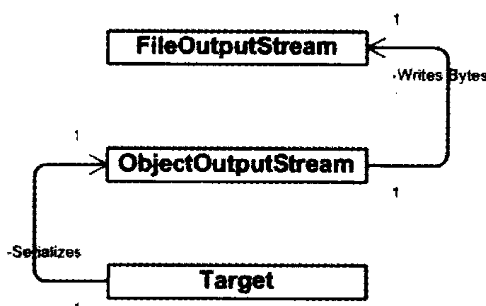
21.5 与装饰模式的关系

在使用串行化或者任何其他的 Java I/O 操作的时候，都不可避免地要使用装饰模式。比如，上面使用过的将一个 TextField 对象串行化的方法如代码清单 3 所示。

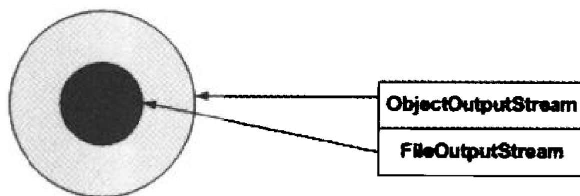
代码清单 3：串行化的源代码

```
private void serializeTextField(
    TextField textField, String filename)
  {
    try
    {
      FileOutputStream outStream =
          new FileOutputStream(filename);
      ObjectOutputStream out =
          new ObjectOutputStream(outStream);
      out.writeObject(textField);
      out.flush();
      out.close();
    }
    catch (Exception e)
    {
      System.out.println(e);
  }
}
```

这个方法使用 ObjectOutputStream 对象将 FileOutputStream 对象包裹起来，以达到将 TextField 对象串行化的需要。串行化过程的结构图如下图所示。



这个装饰过程可以用下图所示的包裹图描述。



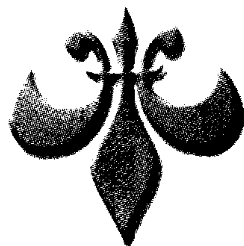
关于装饰模式的细节，请见本书的“装饰（Decorator）模式”一章。

本章介绍了 Java 语言对“冷藏”和“解冻”的处理方法，分析了串行化过程中的装饰模式的应用。本章是对“原始模型（Prototype）模式”一章的有益补充。

参考文献

[HAROLD99] Elliotte Rusty Harold. Java I/O. published by O'Reilly，March 1999

[UENO01] Ken Ueno and the WebSphere Consulting Team. WebSphere V3.5 Handbook. published by Prentice Hall PTR，January 2001



第四部分　结构模式

结构模式（Structural Pattern）描述如何将类或者对象结合在一起形成更大的结构。结构模式描述两种不同的东西：类与类的实例。根据这一不同，结构模式可以分为类的结构模式和对象的结构模式两种。

- 类的结构模式　类的结构模式使用继承来把类、接口等组合在一起，以形成更大的结构。当一个类从父类继承并实现某接口时，这个新的类就把父类的结构和接口的结构结合起来。类的结构模式是静态的。一个类的结构模式的典型例子，就是类形式的适配器模式。
- 对象的结构模式　对象的结构模式描述怎样把各种不同类型的对象组合在一起，以实现新的功能的方法。对象的结构模式是动态的。

一个典型的对象的结构模式就是代理人模式，其他的例子包括后面将要介绍的合成模式、享元模式、装饰模式以及对象形式的适配器模式等。

有一些模式会有类形式和对象形式两种，成为以上两种形式的结构模式的极好注解。本节要介绍的适配器模式就是这样，它有类形式和对象形式两种。

本书将要介绍的结构模式有下面这些：适配器模式、缺省适配模式、合成模式、装饰模式、代理模式、享元模式、门面模式、桥梁模式等。

此外，与适配器模式同时介绍的还有“专题：适配器模式与XMLProperties”；与装饰模式一同介绍的还有“专题：设计模式在Java I/O库中的应用”一章；与代理模式一同介绍的还有“专题：智能引用代理”和“专题：虚拟代理的例子”等。

第 22 章　适配器（Adapter）模式

适配器模式（Adapter Pattern）[GOF95]把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

22.1　引　　言

名称的由来

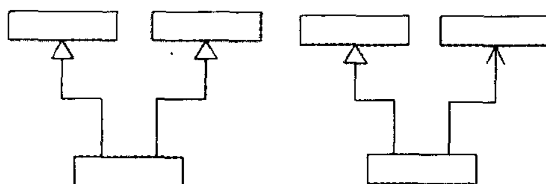
这很像变压器（Adapter），变压器把一种电压变换成另一种电压。把美国的电器拿回中国大陆来用的时候，用户就面临电压不同的问题。美国的生活用电电压是 110V，而中国的电压是 220V。如果要在中国使用在美国的电器，就必须有一个能把 220V 电压转换成 110V 电压的变压器。而这正像是本模式所做的事，因此此模式也常常被称为变压器模式。

读者可能也会想到，Adapter 在中文中也可翻译为转换器（适配器）。实际上，转换器（适配器）也是一个合适的名字。仍用电器做例子，美国的电器的插头一般是三相的，即除了阳极、阴极外，还有一个地极。中国的建筑物内的电源插座一般只有两极，没有地极。这时候，即便电器的确可以接收 220V 电压，电源插座和插头不匹配也使得电器无法使用。一个三相到两相的转换器（适配器）就能解决这个问题。因此，本书将此模式称为适配器模式。

同时，这种做法也很像货物的包装过程：被包装的货物的真实样子被包装所掩盖和改变，因此有人把这种模式叫做包装（Wrapper）模式。事实上，大家经常写很多这样的 Wrapper 类，把已有的一些类包装起来，使之能有满足需要的接口。

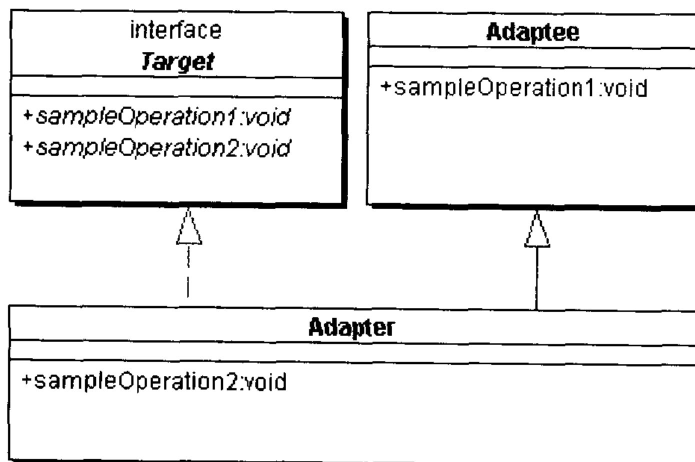
适配器模式的两种形式

适配器模式有类的适配器模式和对象的适配器模式两种不同的形式。如下图所示，左边是类的适配器模式，右边是对象的适配器模式。



22.2 类的适配器模式的结构

类的适配器模式把被适配的类的 API 转换成为目标类的 API，其静态结构图如下图所示。



在上图中可以看出，Adaptee 类并没有 sampleOperation2()方法，而客户端则期待这个方法。为使客户端能够使用 Adaptee 类，提供一个中间环节，即类 Adapter，把 Adaptee 的 API 与 Target 类的 API 衔接起来。Adapter 与 Adaptee 是继承关系，这决定了这个适配器模式是类的。

模式所涉及的角色有：

- 目标（Target）角色：这就是所期待得到的接口。注意，由于这里讨论的是类适配器模式，因此目标不可以是类。
- 源（Adaptee）角色：现有需要适配的接口。
- 适配器（Adapter）角色：适配器类是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

本模式的示意性源代码如代码清单 1 所示。

代码清单 1：Target 的源代码

```
package com.javapatterns.adapter.classAdapter;
public interface Target
{
    /**
     * 这是源类也有的方法 sampleOperation1
     */
    void sampleOperation1();
    /**
     * 这是源类没有的方法 sampleOperation2
     */
```

```
    void sampleOperation2();
}
```

上面给出的是目标角色的源代码，这个角色是以一个 Java 接口的形式实现的。可以看出，这个接口声明了两个方法：sampleOperation1()和 sampleOperation2()。而源角色 Adaptee 是一个具体类，它有一个 sampleOperation1()方法，但是没有 sampleOperation2()方法，如代码清单 2 所示。

代码清单 2：Adaptee 的源代码

```
package com.javapatterns.adapter.classAdapter;
public class Adaptee
{
    /**
     * 源类含有方法 sampleOperation1.
     */
    public void sampleOperation1(){}
}
```

适配器角色 Adapter 扩展了 Adaptee，同时又实现了目标接口。由于 Adaptee 没有提供 sampleOperation2()方法，而目标接口又要求这个方法，因此适配器角色 Adapter 实现了这个方法，如代码清单 3 所示。

代码清单 3：Adapter 的源代码

```
package com.javapatterns.adapter.classAdapter;
public class Adapter
    extends Adaptee implements Target
{
    /**
     *  由于源类没有方法 sampleOperation2,
     *  因此适配器类补充上这个方法
     */
    public void sampleOperation2()
    {
        // Write your code here
    }
}
```

22.3　类的适配器模式的效果

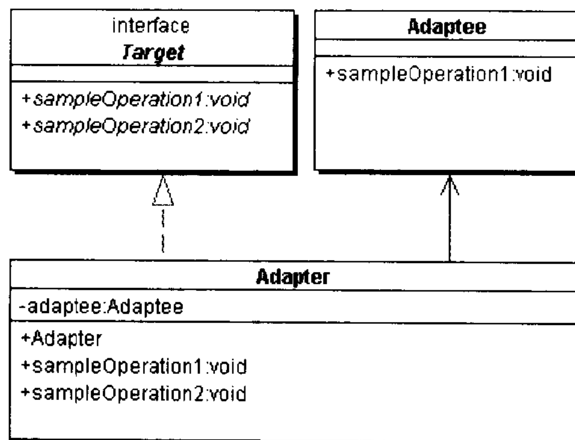
使用一个具体类把源（Adaptee）适配到目标（Target）中。这样一来，如果源以及源的子类都使用此类适配，就行不通了。

由于适配器类是源的子类，因此可以在适配器类中置换掉（Override）源的一些方法。

由于只引进了一个适配器类，因此只有一个路线到达目标类，使问题得到简化。

22.4 对象的适配器模式的结构

与类的适配器模式一样，对象的适配器模式把被适配的类的 API 转换成为目标类的 API，与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到 Adaptee 类，而是使用委派关系连接到 Adaptee 类。对象的适配器模式的静态结构如下图所示。



从上图中可以看出，Adaptee 类并没有 sampleOperation2()方法，而客户端则期待这个方法。为使客户端能够使用 Adaptee 类，需要提供一个包装（Wrapper）类 Adapter。这个包装类包装了一个 Adaptee 的实例，从而此包装类能够把 Adaptee 的 API 与 Target 类的 API 衔接起来。Adapter 与 Adaptee 是委派关系，这决定了这个适配器模式是对象的。

从上图中可以看出，模式所涉及的角色有：

- 目标（Target）角色：这就是所期待的接口，目标可以是具体的或抽象的类。
- 源（Adaptee）角色：现有需要适配的接口。
- 适配器（Adapter）角色：适配器类是本模式的核心。适配器把源接口转换成目标接口，显然，这一角色必须是具体类。

本模式的示意性源代码如代码清单 4 所示。

代码清单 4：Target 的源代码

```
package com.javapatterns.adapter;
public interface Target
{
    /**
     * 这是源类也有的方法 sampleOperation1.
     */
    void sampleOperation1();
    /**
     * 这是源类没有的方法 sampleOperation2.
     */
    void sampleOperation2();
```

```
}
```

上面给出的是目标角色的源代码，这个角色是以一个 Java 接口的形式实现的。可以看出，这个接口声明了两个方法：sampleOperation1()和 sampleOperation2()。而源角色 Adaptere 是一个具体类，它有一个 sampleOperation1()方法，但是没有 sampleOperation2()方法，如代码清单 5 所示。

代码清单 5：Adaptee 的源代码

```
package com.javapatterns.adapter.classAdapter;
public class Adaptee
{
    /**
     * 源类有方法 sampleOperation1.
     */
    public void sampleOperation1(){}
}
```

适配器类的源代码如代码清单 6 所示。

代码清单 6：Adapter 的源代码

```
package com.javapatterns.adapter;
public class Adapter implements Target
{
    private Adaptee adaptee;
    public Adapter(Adaptee adaptee)
    {
        super();
        this.adaptee = adaptee;
    }
    /**
     * 源类有方法 sampleOperation1
     * 因此适配器类直接委派即可
     */
    public void sampleOperation1()
    {
        adaptee.sampleOperation1();
    }
    /**
     * 源类没有方法 sampleOperation2
     * 因此由适配器类需要补充此方法
     */
    public void sampleOperation2()
    {
        // Write your code here
    }
}
```

最后需要指出的是，适配器模式的用意是将接口不同而功能相同或者相近的两个接口加以转换，这里面包括适配器角色补充了一个源角色没有的方法。但是目标接口需要的方法，正如本节在上面给出的例子里，适配器角色实际上补充了一个源角色没有的方法——sampleOperation2()。读者不要误以为适配器模式就是为补充源角色没有的方法而准备的。

对象的适配器模式的效果：

（1）一个适配器可以把多种不同的源适配到同一个目标。换言之，同一个适配器可以把源类和它的子类都适配到目标接口。

（2）与类的适配器模式相比，要想置换源类的方法就不容易。如果一定要置换掉源类的一个或多个方法，就只好先做一个源类的子类，将源类的方法置换掉，然后再把源类的子类当做真正的源进行适配。

（3）虽然要想置换源类的方法不容易，但是要想增加一些新的方法则方便得很，而且新增加的方法可同时适用于所有的源。

22.5 在什么情况下使用适配器模式

在以下各种情况下使用适配器模式：

（1）系统需要使用现有的类，而此类的接口不符合系统的需要。

（2）想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。这些源类不一定有很复杂的接口。

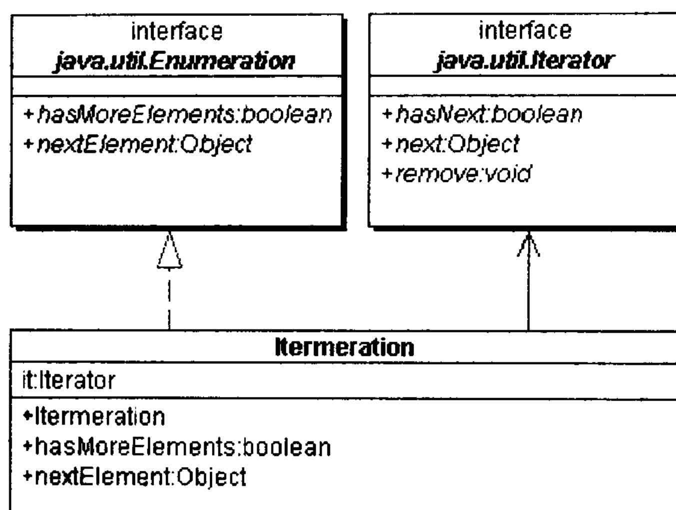
（3）（对对象的适配器模式而言）在设计里，需要改变多个已有的子类的接口，如果使用类的适配器模式，就要针对每一个子类做一个适配器类，而这不太实际。

22.6 Iterator 与 Enumeration

在 JDK 1.0 和 1.1 版本里没有 Java 聚集（Collection）的框架，这一框架是在 JDK 1.2 版本中给出的。与此相对应，JDK 1.0 和 1.1 版本里提供了 Enumeration 接口，而 JDK 1.2 版本给出了 Iterator 接口。如果读者有很多的 Java 代码是为老版本 Java 编译器写的，使用的是 Enumeration，现在想使用新版本编译器和新的 Java 聚集库包的话，需要将已有代码的 Iterator 接口转换成 Enumeration 接口。因为 Java 聚集要求 Iterator 接口，只有这样才能使已有的代码可以使用新版本的聚集对象。

从 Iterator 到 Enumeration 的适配

这就是本书叫做 Itermeration 的类，它使用了对象的适配器模式，将一个 Iterator 对象封装在一个 Enumeration 类型的具体类里面。Itermeration 这个字是英文字 Iterator 的前半截和 Enumeration 的后半截混合而成，如下图所示。



Itermeration 类的源代码如代码清单 7 所示。

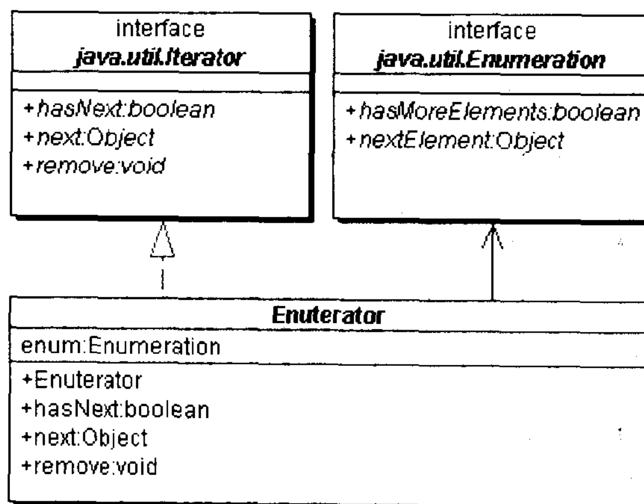
代码清单 7：Itermeration 的源代码

```
package com.javapatterns.adapter.iterenum;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Enumeration;
public class Itermeration
    implements Enumeration
{
    private Iterator it;
   /**
    *  构造子
    */
    public Itermeration(Iterator it)
    {
        this.it = it;
    }
   /**
    *  是否存在下一个元素
    */
    public boolean hasMoreElements()
    {
        return it.hasNext();
    }
   /**
    *  返还下一个元素
    */
    public Object nextElement()
        throws NoSuchElementException
    {
        return it.next();
```

```
    }
}
```

从 Enumeration 到 Iterator 的适配

反过来，Enuterator 使用了对象的适配器模式将 Enumeration 接口适配到 Iterator 接口。Enuterator 这个字是英文字 Enumeration 的前半截和 Iterator 的后半截混合而成。Enuterator 使用了对象的适配器模式，如下图所示。



Enuterator 的源代码如代码清单 8 所示。

代码清单 8：Enuterator 类的源代码

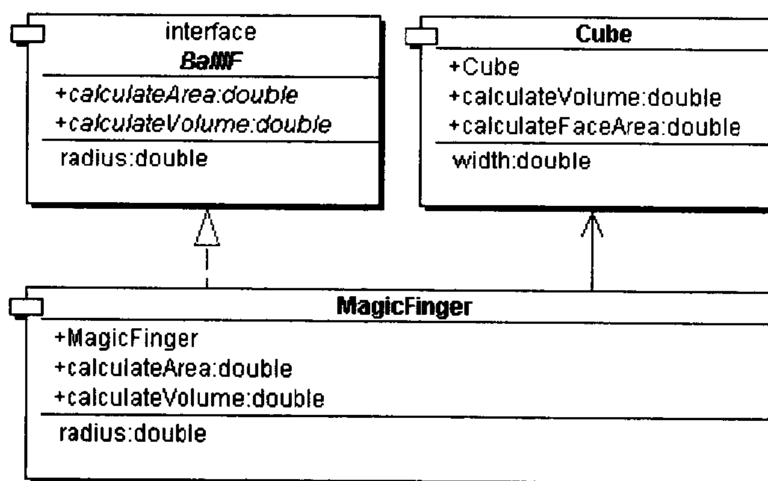
```
package com.javapatterns.adapter.iterenum;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Enumeration;
public class Enuterator
    implements Iterator
{
    Enumeration enum;
   /**
    *  构造子
    */
    public Enuterator(Enumeration enum)
    {
        this.enum = enum;
    }
   /**
    *  是否存在下一个元素
    */
    public boolean hasNext()
    {
```

```
            return enum.hasMoreElements();
        }
        /**
         *  返还下一个元素
         */
        public Object next()
            throws NoSuchElementException
        {
            return enum.nextElement();
        }
        /**
         *  删除当前的元素（不支持）
         */
        public void remove()
        {
            throw new UnsupportedOperationException();
        }
    }
```

22.7　利用适配器模式指方为圆

中国古代有赵高指鹿为马的故事，而《楚辞·九辩》中说："圜凿而方枘兮，吾固知其鉏鋙而难入。"不论是指鹿为马还是指方为圆，都需要适配器模式。本节就是要使用适配器模式这个魔术手指"指方为圆"。

适配器模式在本例子中的类图如下图所示。



首先来看一看 Cube 类的源代码，如代码清单 9 所示，这是类的适配器模式的源角色。

代码清单 9：Cube 类的源代码

```
package com.javapatterns.adapter.cube2ball;
public class Cube
```

```
{
    private double width;
    /**
     *  构造子
     */
    public Cube(double width)
    {
        this.width = width;
    }
    /**
     *  计算体积
     */
    public double calculateVolume()
    {
        return width * width * width;
    }
    /**
     *  计算面积
     */
    public double calculateFaceArea()
    {
        return width * width;
    }
    /**
     *  长度的取值方法
     */
    public double getWidth()
    {
        return this.width;
    }
    /**
     *  长度的赋值方法
     */
    public void setWidth(double width)
    {
        this.width = width;
    }
}
```

下面给出的是目标接口角色，即 BallIF 接口的源代码，如代码清单 10 所示。

代码清单 10：BallIF 接口的源代码

```
package com.javapatterns.adapter.cube2ball;
public interface BallIF
{
    /**
     *  计算面积
```



```
     */
     double calculateArea();
    /**
     *  计算体积
     */
     double calculateVolume();
    /**
     *  半径的取值方法
     */
     double getRadius();
    /**
     *  半径的赋值方法
     */
     void setRadius(double radius);
}
```

MagicFinger 类扮演的是适配器角色，它的源代码如代码清单 11 所示。

代码清单 11：MagicFinger 类的源代码

```
package com.javapatterns.adapter.cube2ball;
public class MagicFinger
    implements BallIF
{
    private double radius = 0;
    private static final double PI = 3.14D;
    private Cube adaptee;
   /**
    *  构造子
    */
    public MagicFinger(Cube adaptee)
    {
        super();

        this.adaptee = adaptee;
        radius = adaptee.getWidth();
    }
   /**
    *  计算面积
    */
    public double calculateArea()
    {
        return PI * 4.0D * ( radius * radius );
    }
   /**
    *  计算体积
    */
   public double calculateVolume()
```

```
    {
        return PI * 4.0D/3.0D *
            ( radius * radius * radius );
    }
    /**
     *  半径的取值方法
     */
    public double getRadius()
    {
        return radius;
    }
    /**
     *  半径的赋值方法
     */
    public void setRadius(double radius)
    {
        this.radius = radius;
    }
}
```

MagicFinger 类实现了 BallIF 接口，并同时扩展了 Cube 类。由于 Cube 类并没有作为 BallIF 所要求的方法，因此 MagicFinger 类提供了默认的方法。这样 MagicFinger 可以替代 BallIF，但同时又具有从 Cube 类继承而来的性能。

如果读者还记得中学数学的话，应该可以看出，这个指方为圆系统其实还是有数学道理的。它接收一个正方体，返还此正方体的内切球，也就是能放进此正方体的最大的球。

显然，本例子里使用的是对象的适配器模式。这样做的好处是，如果一旦决定不仅要支持正方体，而且要支持四面体等多面体，使可以使用同一个 MagicFinger 类，而不必针对每一个多面体都建立一个 MagicFinger 类。

22.8 适配器模式在架构层次上的应用

适配器模式可以在架构层次上使用。

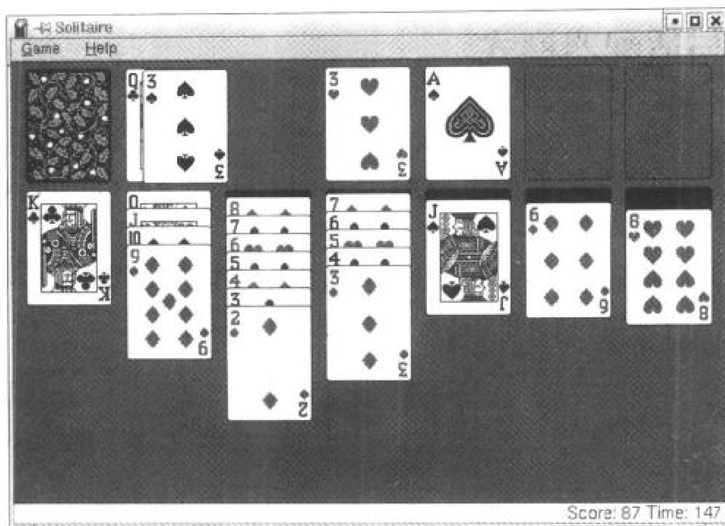
一个例子：WINE

WINE 是一个 Open Source 的免费软件（http://wine.codeweavers.com/），它允许用户在 Linux 环境里运行 Windows 程序。比如下面的 Linux 命令会运行 Windows 的 Solitaire 纸牌程序：

```
wine /data1/dosc/windows/sol.exe
```

Windows 的纸牌游戏 Solitaire（sol.exe）未经修改便在 Linux 上运行的情况如下图所示。





可以看出，WINE 提供了从 Linux 到 Windows 图形界面的适配器，它使得 Windows 的图形界面可以在 Linux 界面里运行，这样就不必修改任何一个图形界面，便可以使两者兼容。

MKS Toolkit 软件

MKS Toolkit 是一个为 WIN 32 系统设计的商业软件（www.mks.com），它提供了 C Shell、Korn Shell、Perl 脚本语言，以及多种 UNIX 命令的解释器。在安装了此软件之后，Windows 的用户可以在自己的系统上使用 Unix 的 C Shell 和 Korn Shell 指令集合，以及如 ls、vi、grep 等的 UNIX 命令。在原则上，一个为 UNIX 系统写的脚本语言可以通过不修改或者很少的修改便可在 WIN 32 系统上运行。

也就是说，MKS Toolkit 提供了 UNIX 命令集从 UNIX 到 Windows 的适配，从而是一种命令集层次上的适配器模式，在某种程度上做的是与 WINE 软件相反的事情。

下面是一个熟知的 UNIX 指令：

```
ls -ltr
```

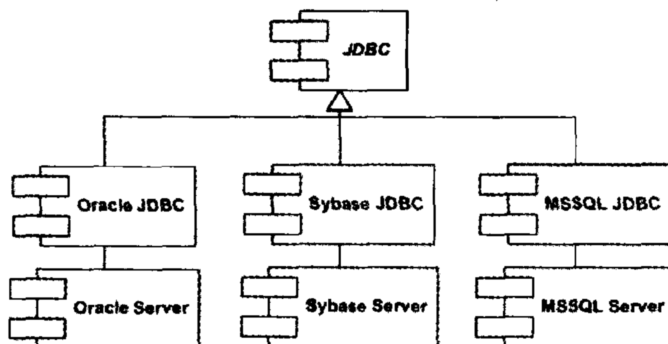
在安装了 MKS 软件之后，可以在 Windows 2000 上运行 UNIX 指令，如下图所示。

```
C:\WINNT\System32\cmd.exe
C:\>ls -ltr
total 2593
-rwxrwxrwa   1 Administrators  None        1323 Sep 24  1997 profile.ksh
-rwxrwxrwa   1 Administrators  None        1942 Sep 24  1997 environ.ksh
drwxrwxrwa   1 Administrators  SYSTEM         0 Dec  9 23:02 Documents a
drwxrwxrwx   1 Administrators  None           0 Dec 10 21:40 Together4.2
drwxrwxrwx   1 Administrators  None           0 Dec 10 22:06 IMNnq_NT
drwxrwxrwx   1 Administrators  None           0 Dec 16 00:38 jdk131
drwxrwxrwx   1 Administrators  None           0 Dec 22 23:39 dpec
drwxrwxrwx   1 Administrators  None           0 Feb 11 16:14 JavaSource
drwxrwxrwx   1 Administrators  None           0 Feb 12 20:55 Tseng
-rwxrwxrwa   1 Administrators  None           0 Feb 27 19:24 AdobeWeb.lo
drwxrwxrwx   1 Administrators  None           0 Mar  3 16:01 Inetpub
dr-xr-xr-x   1 Administrators  None           0 Mar  5 21:58 Acrobat5.0
drwxrwxrwx   1 Administrators  None           0 Mar  9 17:31 mks
drwxrwxrwx   1 Administrators  SYSTEM         0 Mar  9 18:02 Perl
```

JDBC 驱动软件与适配器模式

Sun Microsystem 在 1996 年公开了 Java 语言的数据库连接工具 JDBC。JDBC 使得 Java 语言程序能够连接到数据库上，并使用 SQL（Structured Query Language）来查询和修改数据和数据定义。

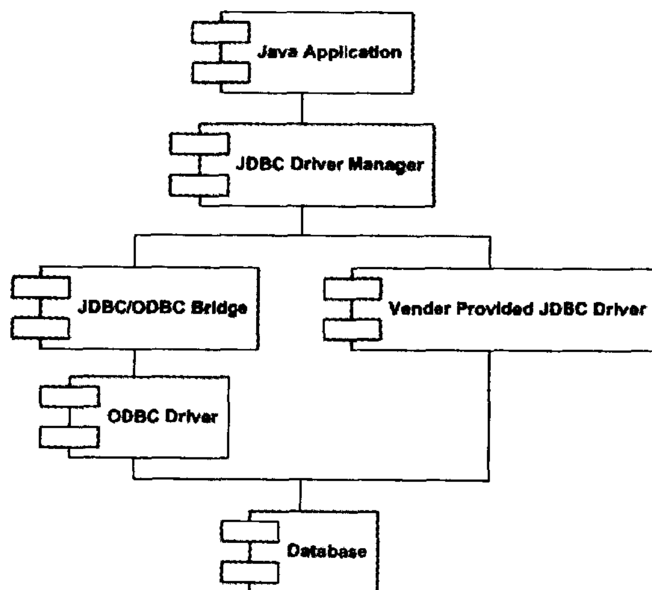
JDBC 给出一个客户端通用的界面。每一个数据库引擎的 JDBC 驱动软件都是一个介于 JDBC 接口和数据库引擎接口之间的适配器软件，如下图所示。



抽象的 JDBC 接口和各个数据库引擎的 API 之间都需要相应的适配器软件，即为各个数据库引擎准备的驱动软件。

JDBC/ODBC 桥梁

JDBC 的想法和微软的 ODBC 想法很接近。只要有合适的驱动软件，JDBC 就可以直接连接到数据库上。如果没有合适的 JDBC 驱动软件，用户也可以通过 ODBC 驱动软件把 JDBC 通过一个 JDBC/ODBC 桥梁软件与 ODBC 驱动软件连接起来，从而达到连接数据库的目的。ODBC/JDBC 桥梁的架构图如下图所示。



JDBC 的库不可能和 ODBC 的库有相同的接口，因此，使用适配器模式将 ODBC 的 API 接口改为 JDBC 的接口就是惟一可行的方法。因此，JDBC/ODBC 桥梁是适配器模式的具体应用。

22.9　关于模式实现的讨论

本模式在实现时有以下这些值得注意的地方：

（1）目标接口可以省略。此时，目标接口和源接口实际上是相同的。由于源是一个接口，而适配器类是一个类（或抽象类），因此这种做法看似平庸而并不平庸，它可以使客户端不必实现不需要的方法。这一点将在缺省适配模式一章里做详尽的分析。

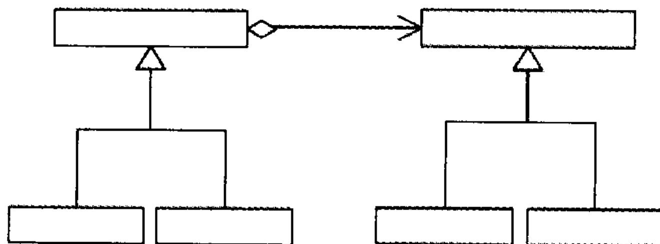
（2）适配器类可以是抽象类。这可以在缺省适配的情况下看到。

（3）带参数的适配器模式。使用这种办法，适配器类可以根据参数返还一个合适的实例给客户端。

22.10　适配器模式与相关的模式

适配器模式与桥梁模式的关系

对于客户端来说，一个适配器类将一个 Adaptee 类的接口改成 Target 的接口。桥梁模式的简略类图如下图所示。

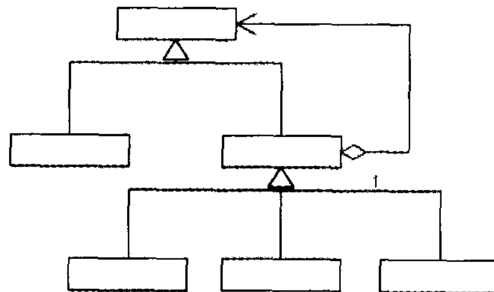


在桥梁模式里面，Abstraction 对象定义了 Implementation 对象的接口，目的是要隐藏在 Abstraction 对象后面的实现细节。Abstraction 对象看上去也像是把客户端的接口与 Implementation 的接口相连接。但是两者有明显的本质上的不同，并且这种不同反映在用意上和应用上。

桥梁模式的用意是要把实现和它的接口分开，以便它们可以独立地变化。桥梁模式并不是用来把一个已有的对象接到不相匹配的接口上的。当一个客户端只知道一个特定的接口，但是又必须与具有不同接口的类打交道时，就应当使用适配器模式。

适配器模式与装饰模式的关系

一个装饰类也是位于客户端和另外一个 Component 对象之间的，在它接到客户端的调用后把调用传给一个或几个 Component 对象。一个纯粹的装饰类必须与 Component 对象在接口上的完全相同，并增强后者的功能。装饰模式的简略类图如下图所示。



与适配器类不同的是，装饰类不能改变它所装饰的 Component 对象的接口。

适配器模式与缺省适配模式的关系

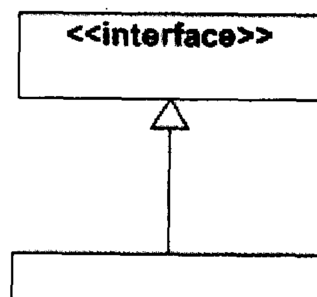
在 JDK1.2 中有很多为事件处理而设计的事件监听适配器类。如 java.awt.event 包中有：

- ComponentAdapter
- ContainerAdapter
- FocusAdapter
- HierarchyBoundsAdapter
- KeyAdapter
- MouseAdapter
- MouseMotionAdapter
- WindowAdapter

而 Javax.swing.event 包中有：

- InternalFrameAdapter
- MouseInputAdapter

缺省适配模式类图如右图所示。



这些 Java 类均是为了使用的方便而提供的缺省适配器接口，它们为所对应的 Java 接口提供空的或称“平庸的”实现。这样，Java 语言的用户程序如果需要实现一个接口中很少的几个方法时，就不必繁琐地给出它所不需要的那些方法的空的实现。

关于缺省适配模式，请见“缺省适配（Default Adapter）模式”，以及“专题：观察者模式与 AWT 中的事件处理”等章。

问答题

1. 请做一个 Kittie 的具体类，并实现 miao()、catchRat()、run()、sleep()等方法。 再做一个 Puppie 的接口，要求有 wao()，fetchBall()，run()，sleep()等方法。

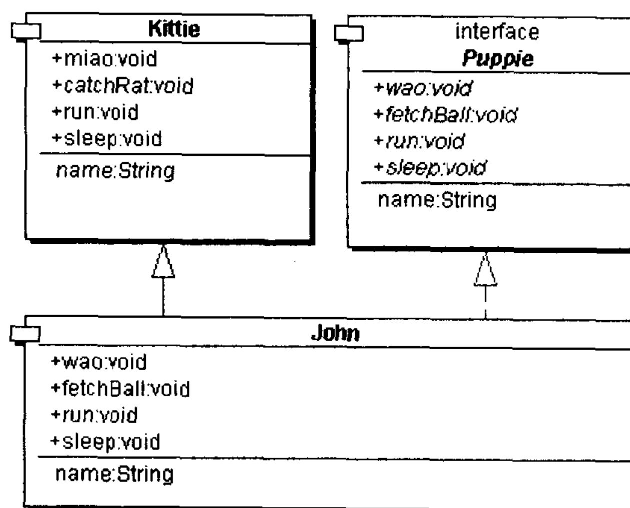
假设现在 John 有一个朋友想要一只 Puppie，可是只找到一只 Kittie。John 需要使用适配器模式把 Kittie “适配” 成 Puppie，让他的朋友满意。请问 John 该怎么做（提示：量力而为）。

2. 请指出第 1 题的解答所使用的是哪一种形式的适配器模式。

3. 写一个简单的 Java 语言程序读取 Excel 文件中的数据。（提示：使用 JDBC/ODBC Bridge）

问答题答案

1. 根据提示可以量力而为，因此 John 需要把 miao() “适配” 成 wao()，catchRat() “适配” 成 fetchBall()，run()，sleep()则保持不变。系统的设计图如下图所示。



系统的源代码如代码清单 12 所示。

代码清单 12：Puppie 类的源代码

```
package com.javapatterns.adapter.kittie2puppie;
public interface puppie
{
    void wao();
    void fetchBall();
    void run();
    void sleep();
    void setName(String name);
    String getName();
}
```

系统需要的是带有 Puppie 接口的类，上面给出的就是扮演目标角色的 Puppie 接口的源代码。下面给出的是扮演适配器模式中的源角色的 Kittie 类的源代码，如代码清单 13 所示。Kittie 类是系统现有的功能相近的一个类，但是它的接口与 Puppie 的接口不同，因此不符合要求。

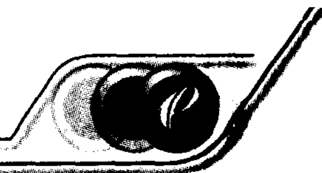
代码清单 13：Kittie 类的源代码

```
package com.javapatterns.adapter.kittie2puppie;
public class Kittie
{
    public void miao(){}
    public void catchRat()
    {
        //write you code here
    }
    public void run()
    {
        //write you code here
    }
    public void sleep()
    {
        //write you code here
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
}
```

适配器模式中的核心角色是适配器角色，这个角色由 John 类扮演，如代码清单 14 所示。

代码清单 14：适配器类的源代码

```
package com.javapatterns.adapter.kittie2puppie;
public class John extends Kittie
    implements puppie
{
    public void wao()
    {
        this.miao();
    }
    public void fetchBall()
    {
```



```
        this.catchRat();
    }
    public void run()
    {
        super.run();
    }
    public void sleep()
    {
        super.sleep();
    }
    public String getName()
    {
        return super.getName();
    }
    public void setName(String name)
    {
        super.setName(name);
    }
}
```

可以看出 John 自己实现了 Puppie 接口，并借用了 Kittie 类的功能。

2. 这里使用的是类的适配器模式。

3. 首先，建立一个 ODBC，连接到所需要的 Excel 文件上。假设这个 ODBC 叫 myExcelFile。下面的 Java 语言程序会读出这个 Excel 文件的第一个 sheet，如代码清单 15 所示。

代码清单 15：读取 Excel 文件的源代码

```
import java.sql.*;
public class ExcelReader
{
    public static String readExcel(
        String ODBCEntry,
        String sheetNumber)
    {
        StringBuffer ret = new StringBuffer(1000);

        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            Connection conn =
                DriverManager.getConnection(
                "jdbc:odbc:" + ODBCEntry );

            Statement stmt = conn.createStatement();
            ResultSet rs =
```

```
                stmt.executeQuery( "Select * from [Sheet"
                + sheetNumber + "$]" );

            ResultSetMetaData rsmeta = rs.getMetaData();
            int numberOfColumns = rsmeta.getColumnCount();

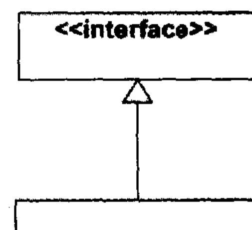
            while (rs.next())
            {
                for (int i = 1; i <= numberOfColumns; i++)
                {
                    if (i   1) ret.append(",");
                    ret.append(rs.getString(i));
                }
                ret.append("\n");
            }
            stmt.close();
            conn.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            return ret.toString();
        }
    }
    public static void main(String[] args)
    {
        System.out.println(readExcel("myExcelFile","1"));
    }
}
```

参考文献

[SAVARESE02] Daniel F. Savarese. What Dynamic Proxies Can Do For You. Java Pro, May 2002

第 23 章　缺省适配（Default Adapter）模式

缺省适配模式为一个接口提供缺省实现，这样子类型可以从这个缺省实现进行扩展，而不必从原有接口进行扩展。作为适配器模式的一个特例，缺省适配模式在 Java 语言中有着特殊的应用。右图所示就是缺省适配模式的简略类图。



在阅读本章之前，请首先阅读本书的“适配器模式”一章，在阅读之后可以参考“专题：观察者模式与 AWT 中的事件处理”一章中关于 Java 事件处理机制的讨论。

23.1 鲁智深的故事

和尚要做什么呢，要吃斋、念经、打坐、撞钟、习武等。如果设计一个和尚接口，给出所有的和尚都需要实现的方法，那么这个接口应当如代码清单 1 所示。

代码清单 1：和尚接口给出所有的和尚必须遵守的操作

```
interface 和尚
{
  public void 吃斋（）;
  public void 念经（）;
  public void 打坐（）;
  public void 撞钟（）;
 public void 习武（）;
 public String getName();
}
```

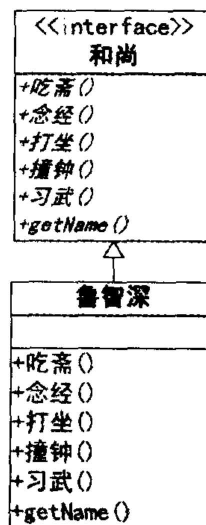
显然，所有的和尚类都应当实现接口所定义的全部方法，不然就根本通不过 Java 语言编译器。像下面的鲁智深类就不行，如代码清单 2 所示。

代码清单 2：鲁智深类只实现了 getName()和习武()方法

```
class 鲁智深 implements 和尚
{
    public void 习武()
    {
        拳打镇关西;
```

```
        大闹五台山;
        大闹桃花村;
        火烧瓦官寺;
        倒拔垂杨柳;
        火烧瓦官寺;
    }
        public String getName()
    {
        return "鲁智深";
    }
}
```

由于鲁智深只实现了 getName()和习武()方法，而没有实现任何其他的方法。因此，它根本就通不过 Java 语言编译器。那么是不是必须像右图所示的那样，实现所有的方法才行呢？



在上面的图中，鲁智深类只有实现所有的方法。这样一来鲁智深就不再是鲁智深了。以史为鉴，可以知天下。研究一下几百年前鲁智深是怎么剃度成和尚的，会对 Java 编程有很大启发。不错，当初鲁达剃度，众僧说："这个人形容丑恶，相貌凶顽，不可剃度他"，但是长老却说："此人上应天星，心地刚直。虽然时下凶顽，命中驳杂，久后却得清净。证果非凡，汝等皆不及他。"

原来如此！看来只要这里也应上一个天星的话，如代码清单 3 所示，问题就可解决了！使用 OO 的语言来说，"应"者，实现也；"天星"者，抽象类也。

代码清单 3：天星实现了所有的和尚接口的方法

```
abstract class 天星 implements 和尚
{
public void 习武(){}
public void 吃斋(){}
public void 念经(){}
public void 打坐(){}
 public void 撞钟(){}
 public String getName()
    {
        return null;
    }
}
```

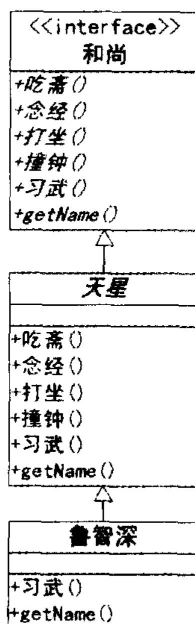
鲁智深的 extends 抽象类如代码清单 4 所示。

代码清单 4：extends 类的源代码

```
class 鲁智深 extends 天星
{
    public void 习武()
```

```
    {
        拳打镇关西;
        大闹五台山;
        大闹桃花村;
        火烧瓦官寺;
        倒拔垂杨柳;
        火烧瓦官寺;
    }
        public String getName()
    {
        return "鲁智深";
    }
}
```

显然，这个鲁智深类可以通过 Java 编译器。这是一个正确的答案。这个答案的类图如下图所示，可以看出，鲁智深类可以只实现它所需要的方法。



这个抽象的天星类便是一个适配器类，鲁智深实际上借助于适配器模式达到了剃度的目的。此适配器类实现了和尚接口所要求的所有方法。但是与通常的适配器模式不同的是，此适配器类给出的所有的方法的实现都是“平庸”的。这种“平庸化”的适配器模式称做缺省适配模式。

23.2 WindowAdapter 是缺省适配模式

在很多情况下，必须让一个具体类实现某一个接口，但是这个类又用不到接口所规定的所有的方法。通常的处理方法是，这个具体类要实现所有的方法，那些有用的方法要有实现，那些没有用的方法也要有空的、平庸的实现。

这些空的方法是一种浪费，有时也是一种混乱。除非看过这些空方法的代码，程序员可能会以为这些方法不是空的。即便他知道其中有一些方法是空的，也不一定知道哪些方法是空的，哪些方法不是空的，除非看过这些方法的源代码或是文档。

比如 WindowListener 接口就定义了七个方法，如代码清单 5 所示。

代码清单 5：接口 WindowListener 的源代码

```
package java.awt.event;
import java.util.EventListener;
public interface WindowListener
    extends EventListener
{
    /**
     * 当视窗第一次变得看得见时
     */
    public void windowOpened(WindowEvent e);
    /**
     * 当用户试图从视窗的系统菜单关闭视窗时
     */
    public void windowClosing(WindowEvent e);
    /**
     * 当视窗已经因调用 dispose()被关闭后
     */
    public void windowClosed(WindowEvent e);
    /**
     * 当视窗被最小化时
     */
    public void windowIconified(WindowEvent e);
    /**
     * 当视窗从最小化状态恢复成正常状态时
     */
    public void windowDeiconified(WindowEvent e);
    /**
     * 当视窗被激活时
     */
    public void windowActivated(WindowEvent e);
    /**
     * 当视窗不再是激活状态时
     */
    public void windowDeactivated(WindowEvent e);
}
```

下面的 Java 小程序就是实现 WindowListener 接口的一个例子，如代码清单 6 所示。可以看出，appletFrame 类只实现了一个有意义的方法，而实现了六个空的方法。

代码清单 6：实现 WindowListener 接口的例子

```
class appletFrame extends Frame
```

```
    implements WindowListener
{
    appletFrame()
    {
        addWindowListener(this);
    }
    public void windowClosed(WindowEvent event)
    {  }
    public void windowDeiconified(WindowEvent event)
    {  }
    public void windowIconified(WindowEvent event)
    {  }
    public void windowActivated(WindowEvent event)
    {  }
    public void windowDeactivated(WindowEvent event)
    {  }
    public void windowOpened(WindowEvent event)
    {  }
    public void windowClosing(WindowEvent event)
    {
        System.out.println("进入 windowClosing");
        this.setVisible(false);
        dispose();
        System.out.println("退出 windowClosing");
        System.exit(0);
    }
}
```

适配器模式可以很好地处理这一情况。可以设计一个抽象的适配器类实现接口，此抽象类要给接口所要求的每一种方法都提供一个空的方法。就像帮助了鲁智深的“上应天星”一样，此抽象类可以使它的具体子类免于被迫实现空的方法。

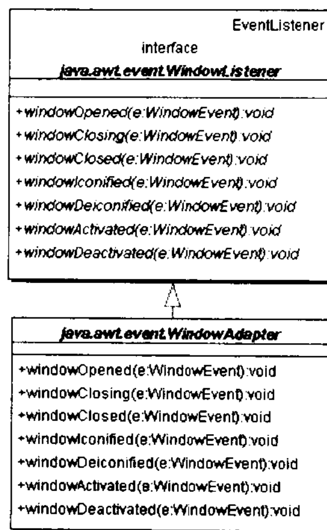
其实，这个抽象类已经有了，它就是 java.awt.WindowAdapter，其源代码如代码清单 7 所示。

代码清单 7：实现 WindowListener 接口的抽象类 WindowAdapter

```
package java.awt.event;
public abstract class WindowAdapter
    implements WindowListener
{
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
```

```
}
```

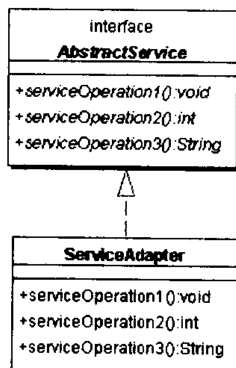
接口 WindowListener 和抽象类 WindowAdapter 的关系图如下图所示。



好了，现在就看一看“平庸”化的适配器模式的结构吧。

23.3　模式的结构

缺省适配模式是一种“平庸”化的适配器模式，这个模式的静态结构可以通过下图所示的类图看清楚。



这个模式的源代码分别给出如下。首先是 Java 接口的源代码，它声明了几个方法，如代码清单 8 所示。

代码清单 8：需要缺省适配的 AbstractService 类的源代码

```
public interface AbstractService
{
    void serviceOperation1();
    int serviceOperation2();
```



```
    String serviceOperation3();
}
```

而具体类的源代码如代码清单 9 所示，这个具体类实现了上面的 Java 接口。

代码清单 9：缺省适配类 ServiceAdapter 的源代码

```
public class ServiceAdapter
    implements AbstractService
{
    public void serviceOperation1(){ }
    public int serviceOperation2(){ return 0; }
    public String serviceOperation3(){ return null; }
}
```

可以看到，接口 AbstractService 要求定义三个方法，分别是 serviceOperation1()、serviceOperation2()和 serviceOperation3()；而抽象适配器类 ServiceAdapter 则为这三种方法都提供了平庸的实现。因此，任何继承自抽象类 ServiceAdapter 的具体类都可以选择它所需要的方法实现，而不必理会其他的不需要的方法。

AWT 库中的 WindowAdapter 类和其他几种类似的 Adapter 类一样，都是缺省适配模式的应用例子。以 WindowAdapter 为例，它是缺省适配模式的实现，看似平庸，而不平庸。

适配器模式的用意是要改变源的接口，以便与目标类的接口相容。缺省适配的用意稍有不同，它是为了方便建立一个不平庸的适配器类而提供的一种平庸实现。

23.4　在什么情况下使用本模式

在任何时候，如果不准备实现一个接口的所有方法时，就可以模仿 WindowAdapter 的做法制造一个抽象类，给出所有方法的平庸的具体实现。这样，从这个抽象类再继承下去的子类就不必实现所有的方法了。这时候，新的类就是 Adapter 模式中的 Adapter 类，而给出平庸实现的抽象类就可以被当做 Adaptee 类。

适配器模式把一个类的接口变换成客户端所期待的另一种接口。适配器模式使原本无法在一起工作的两个类能够在一起工作。如前所述，适配器模式的“平庸化”形式可以使所考察的类不必实现不需要的那部分接口。

23.5　模式的实现

缺省适配模式的中心是一个缺省适配类。这个类应当是抽象类，因为这个类不应当实例化，它的实例也没有用处。

但是缺省适配类所提供的方法却应当是具体的方法，而不是抽象的方法，因为按照模式的用意，这些方法之所以存在，就是为了提供默认实现，以便缺省适配类的具体子类可以按照需要只实现需要实现的方法，忽略不需要实现的方法。

23.6 J2SE 中的缺省适配模式

在 Java 2.0 的 java.awt 库中，除了 WindowAdapter 之外，还有很多的缺省适配类，如下表所示。

适配器类	事件监听器接口
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

这些都是缺省适配模式使用的实际例子，读者可以参考 AWT 中的各种事件监听适配器类的类图，如下图所示。

在前面已经给出了 WindowAdapter 类的源代码。下面给出 ComponentAdapter 的源代码，如代码清单 10 所示。

代码清单 10：ComponentAdapter 的源代码

```
package java.awt.event;
public abstract class ComponentAdapter
    implements ComponentListener
{
  /**
   * 当构件的大小发生变化时被激发
   */
  public void componentResized(ComponentEvent e) {}

  /**
   * 当构件的位置发生变化时被激发
   */
  public void componentMoved(ComponentEvent e) {}
      /**
```



```
 * 当构件变得可见时被激发
 */
public void componentShown(ComponentEvent e) {}
    /**
 * 在构件变成不可见时被激发
 */
public void componentHidden(ComponentEvent e) {}
}
```



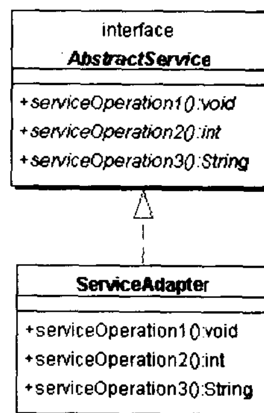
ComponentAdapter 类中的方法全都是空的。这使得它的子类可以选择任何子类所需要的方法置换掉即可，而不需要在子类中给出很多空的方法。如果这些方法被声明为 abstract 的话，子类就不得不为所有的方法给出实现，哪怕是空的实现。

通常使用 ComponentAdapter 类型的对象的方法是把它当做一个 WindowListener 类型的对象登记到一个视窗组件上。当这个视窗组件发生大小变化或位置移动的事件时，这个 ComponentAdapter 对象的相应的方法就会受到激发。当然 ComponentAdapter 类并没有任何有意义的行为。

如果想在事件发生时激发有意义的行为的话，就需要设计一个 ComponentAdapter 类的子类，置换掉相应的方法，并用这个子类代替 ComponentAdapter 类登记到视窗组件上。这样，当事件发生时，相应的方法就会被激发，产生非平庸的行为。

23.7 一个例子

如上所述，抽象类 WindowAdapter 是为接收视窗的事件而准备的。此抽象类内所有的方法都是空的。使用此类可以很方便地创立 listener 对象，置换掉（Override）感兴趣的那个事件所对应的方法。如果不使用此抽象类，那么必然要实现 WindowsListener 接口，而那样就不得不实现所有接口中的方法，即便是不需要的事件所对应的方法，也要给出一个空的方法，而这显然不方便。



显然，抽象类 WindowAdapter 的目标接口可以选得与源接口一样，而不影响效果。这就解释了为什么目标接口不出现在 WindowAdapter 类图里，如右图所示。

SwingUI 类的源代码如代码清单 11 所示。

代码清单 11：SwingUI 类的源代码

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
class SwingUI extends JFrame
```

```
    implements ActionListener
{
    JLabel text, clicked;
    JButton button, clickButton;
    JPanel panel;
    private boolean m_clickMeMode = true;
    Public SwingUI()
    {
      text = new JLabel("我很高兴！ ");
      button = new JButton("理我");
      button.addActionListener(this);
      panel = new JPanel();
      panel.setLayout(new BorderLayout());
      panel.setBackground(Color.white);
      getContentPane().add(panel);
      panel.add(BorderLayout.CENTER, text);
      panel.add(BorderLayout.SOUTH, button);
    }
    public void actionPerformed(ActionEvent event)
    {
          Object source = event.getSource();
          if (m_clickMeMode)
          {
            text.setText("我很烦！ ");
            button.setText("别理我");
            m_clickMeMode = false;
          }
          else
          {
            text.setText("我很高兴！ ");
            button.setText("理我");
            m_clickMeMode = true;
          }
    }
    public static void main(String[] args)
    {
      SwingUI frame = new SwingUI();
      frame.setTitle("我");

      WindowListener listener = new WindowAdapter()
      {
        public void windowClosing(WindowEvent e)
        {
          System.exit(0);
        }
      };
```

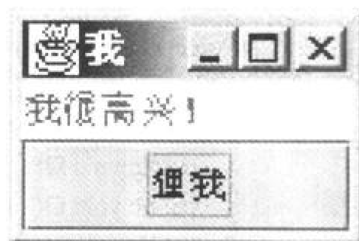


```
        frame.addWindowListener(listener);
        frame.pack();
        frame.setVisible(true);
    }
}
```

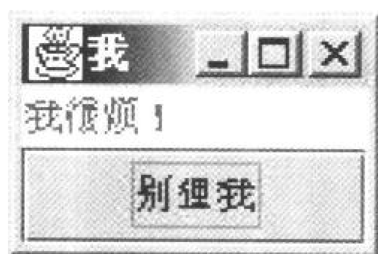
红色的代码就是使用 WindowAdapter 的无名内部类。

显然，由于无名内部类是继承自 WindowAdapter 抽象类，因此只需置换掉（Override）感兴趣的 WindowClosing()方法而不必操心 WindowListener 的其他方法。

SwingUI 运行时的样子如右图所示。



单击命令键“理我”就变成下图所示的样子。



再单击命令键“别理我”就会回到前面一个图的样子。

问答题

1. 假设你有一个接口，这个接口有五个方法：

```
interface InterfaceA {
    public void f1();
    public void f2();
    public void f3();
    public void f4();
    public void f5();
}
```

你准备写一个 ClassB 实现这个接口，但是你只对接口所规定的五个方法中的一个即方法 f1()感兴趣。如果你把 ClassB 写成下面这样：

```
class ClassB implements InterfaceA
{
    public void f1() {/*...*/}
}
```

请问行不行？如果不行，请给出一个可行的解决方案。



使用缺省适配模式。

问答题答案

1. 这样不行，Java 语言编译器会给出编译错误，因为 ClassB 类没有实现接口 InterfaceA 的所有的方法。可行的解决方案是使用适配器模式，首先设计一个抽象类如下所示：

```
abstract class AdapterC implements InterfaceA
{
    public void f1() { }
    public void f2() { }
    public void f3() { }
    public void f4() { }
    public void f5() { }
}
```

然后让 ClassB 继承自抽象类 AdapterC：

```
class ClassB extends AdapterC
{
    public void f1() {/*...*/}
}
```

这样就解决了问题。

参考文献

[GMCC00]Glen McCluskey. Java Developer ConnectionSM (JDC) Tech Tips. http://developer.java.sun.com/developer/TechTips/2000/tt0509.html，May 09，2000

[GVOSS98]Greg Voss. Java Beans Tutorial. Part 4 Introduction，JavaSoft (January 1998)，http://developer.java.sun.com/developer/onlineTraining/Beans/Beans4/index.html#TOC

第 24 章　专题：XMLProperties 与适配器模式

请读者在阅读本章之前，首先阅读本书的“适配器（Adapter）模式”一章。

24.1 引　　言

问题

几乎所有的应用程序都有一些系统常量需要存储在程序外部。一个常见的做法是使用一个配置文件（Configuration File）存储这些常量。这样一旦这些常量有变化的话，无需修改程序代码就可以重新配置一个系统。

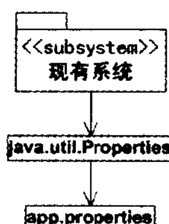
配置文件

在 WIN 31 编程里，一个配置文件就是.ini 文件或者系统 Registry。在 Java 编程里，一个配置文件就是.properties 文件或者 XML 文件。使用.properties 文件虽然是现行工业标准，但正在渐渐被 XML 文件所取代。一个配置文件所需要的无非就是存储一些代码和代码所代表的值组成的对，即 key-value 对。.properties 配置文件的内容如代码清单 1 所示。

代码清单 1：一个.properties 配置文件

```
name=webendshere
website=www.webendshere.com
email=webmaster@webendshere.com
techsupport=support@webendshere.com
```

一个现有的使用 app.properties 配置文件的系统如下图所示。



如果将此配置文件改写成 XML 文件的话，应当如代码清单 2 所示。

代码清单 2：一个.xml 配置文件

```
<property name="name">
    webendshere
<property>
<property name="website">
    www.webendshere.com
<property>
<property name="email">
    webmaster@webendshere.com
<property>
<property name="techsupport">
    support@webendshere.com
<property>
```

使用 XML 文件是一个好主意，惟一的不足是读取 XML 文件不像读取.properties 文件那样容易，因为 Java 语言为.properties 文件准备了 java.util.Properties 类。那么可不可以自己提供一个读取 XML 配置文件的工具类呢？

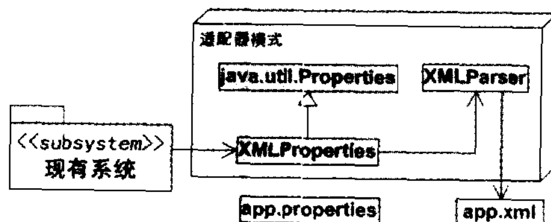
扩展而不修改

当然可以自己提供一个读取 XML 配置文件的工具类。如果是单纯读取 XML 文件的话，事情并不复杂。但是有一些已经开发完成的系统已经使用了.properties 文件，将.properties 文件更换为.xml 文件并不困难，但是更改已有的代码则是很麻烦的事，因为它必然要涉及到修改和重新检测整个已有的系统。

正如本书在前面所述，软件系统的第一条设计原则就是所谓的“开–闭”原则，说的是一个软件系统应当对扩展开放，而对修改封闭。那么能否在将.properties 文件更换为.xml 文件的同时尽量不修改代码呢？

使用适配器模式

如果使用适配器模式的话，是可以做到这一点的。假设这个类叫做 XMLProperties，那么 XMLProperties 需要能读取 XML 文件，但是提供给客户端的是一个 Properties 类型。换言之，XMLProperties 将 XML 接口适配成 java.util.Properties 类型。一个现有的可以使用 app.xml 配置文件的系统，并利用适配器模式作为对现有系统的扩展，如下图所示。



为了使不熟悉 SAX2 的读者能够读懂本节，本书在这里提供一个针对 Java 程序员的



快速 XML 和 SAX2 的教程。关于 SAX2 的事件处理请参阅本书的“专题：观察者模式与 SAX2 浏览器”一章。

24.2　SAX2 浏览器

SAX2 是什么

SAX 是 Simple API for XML 的缩写，它是一个通用的为各种不同的 XML 浏览器而设计的接口，正如 JDBC 是为各种不同的关系数据库而设计的接口一样。使用 SAX 接口而不是直接使用 XML 浏览器本身，可以避免使应用程序依赖于这种 XML 浏览器。

它虽然不是由 W3C 所提出的标准，但却是事实上的工业标准，而且几乎所有的 XML 浏览器都会支持它。与 DOM 比较而言，SAX 是一种轻量型的方法。在处理 DOM 时需要读入整个 XML 文件，然后在内存中创建 DOM 的树结构，包括生成 DOM 树上的每个节点对象。当文件比较小的时候，这不会造成什么问题，但是一旦文件大起来，处理 DOM 就会耗费相当大的资源，以至于在某些应用中变得不实际。这时候，一个很好的替代方法就是使用 SAX。

不同于 DOM 的文档驱动，SAX 是事件驱动的。也就是说，它并不需要读入整个文档，而文档的读入过程也就是 SAX 的解析过程。所谓事件驱动，就是 Java 1.1 新提出的委派事件模型机制（简称为 DEM）。

XMLReader 读入文档的过程和解析的过程是同时进行的。解析开始之前，需要向 XMLReader 注册一个 ContentHandler，也就是相当于一个事件监听器，在 ContentHandler 中定义了很多事件方法，作为对遇到一个 XML 标识符事件的相应方法。比如 startDocument()，它定义了在解析过程中遇到文档开始时采取的操作。当 XMLReader 读到合适的内容时，就会抛出相应的事件，并把这个事件的处理权委派给 ContentHandler，而后者则调用相应的方法作为响应。

SAX2 的运行环境

SAX2 作为 SAX 的第二版，比较起第一版来有很大的改进和不同。要想使用 SAX2，必须要有如下的环境：

- Java 1.1 或更高版本的 JDK
- 一个与 SAX2 兼容的 XML 浏览器（如 xerces）安装在 classpath 上
- SAX2 安装在 classpath 上

怎样浏览 XML 文件

首先，需要一个继承自 DefaultHandler 的类，如代码清单 3 所示。

代码清单 3：MySAXApp 类的源代码

```
package com.javapatterns.xmlproperties;
import org.xml.sax.helpers.DefautlHandler;
public class MySAXApp extends DefaultHandler
{
    public MySAXApp(){ super(); }
}
```

接下来加上一个静态的 main 方法，并在此方法里调用 XMLReaderFactory 类的 createXMLReader()方法。这样一来，MyXMLApp 类就变成代码清单 4 所示。

代码清单 4：MyXMLApp 类的源代码

```
package com.javapatterns.xmlproperties;
import org.xml.sax.helpers.DefautlHandler;
import org.xml.sax.helpers.XMLReaderFactory;
public class MySAXApp extends DefaultHandler
{
    public MySAXApp(){ super(); }

    public static void main(String[] args)
        throws Exception
    {
        XMLReader xr =
            XMLReaderFactory.createXMLReader();
    }
}
```

为了设定 Java 环境的默认值，需要对 Java JVM 指定参量 org.xml.sax.driver 的值，也就是一个 SAX 驱动类的名字。作者所使用的驱动软件包是 Xerces，因此，应当告诉 Java 的 JVM 所用的驱动类名是 org.apache.xerces.parsers.SAXParser，如代码清单 5 所示。

代码清单 5：告诉 JVM 使用正确的驱动类

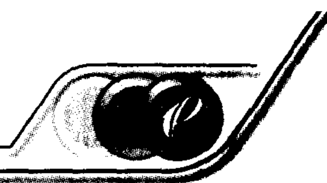
```
java -Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser MySAXApp myxml.xml
```



myxml.xml 是任何想浏览的 XML 文件。

一些流行的驱动软件包如下表所示。

驱动类的名字	注　解
org.apache.xerces.parsers.SAXParser	开放源代码，有合法性检验功能
org.apache.crimson.parser.XMLReaderImpl	开放源代码，有合法性检验功能
oracle.xml.parser.v2.SAXParser	有合法性检验功能
gnu.xml.aelfred2.XmlReader	有合法性检验功能，免费
gnu.xml.aelfred2.SAXDriver	轻体浏览器，没有合法性检验功能，免费



当然，如果不在意把应用程序绑定在某一种驱动软件包上，那么也可以直接调用此驱动类的构造子。同样，假定所使用的驱动软件包是 Xerces，则可以直接使用，如代码清单 6 所示。

代码清单 6：MySAXApp 类的源代码

```
public static void main(String[] args) throws Exception
{
    XMLReader xr = new
        org.apache.xerces.parsers.SAXParser();
}
```

直接调用 driver 的构造子，这样一来程序将被绑定在这种 driver 上。

不把程序绑定在任何一种驱动软件包上的做法的好处是，一旦想要更换 XML 浏览器的驱动类，则只需要把 JVM 里的环境变量 org.xml.sax.driver 改变过来即可。这样做不好的地方是在一个 JVM 中不能同时使用两个不同的驱动软件包。

在使用 MySAXApp 类浏览 XML 文件之前，还必须向事件处理器登记感兴趣的事件。

登记事件处理器

使用 XMLReader 接口的 setContentHandler 和 setErrorHandler 方法可以登记事件处理器，这样一来 XMLReader 在浏览过程中发生某个事件时，XMLReader 会自动激发所登记的事件处理器，如代码清单 7 所示。

代码清单 7：MySAXApp 类被登记成事件处理器

```
public static void main(String[] args)
    throws Exception
{
    XMLReader xr =
        XMLReaderFactory.createXMLReader();
    MySAXApp handler = new MySAXApp();
    xr.setContentHandler(handler);
    xr.setErrorHandler(handler);
}
```

以上的源代码把 MySAXApp 的一个实例登记到 XMLReader 里，以接收正常的内容事件和出错事件。还有其他的一些事件是本项目不需要的，也就没有登记。

如同代码清单 5 中所示，需要浏览的 XML 文件名是通过命令行传进程序的，如代码清单 8 所示。

代码清单 8：MySAXApp 类读入所指定的 XML 文件

```
public static void main(String[] args)
    throws Exception
{
    XMLReader xr = XMLReaderFactory.createXMLReader();
    MySAXApp handler = new MySAXApp();
```

```
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);

        FileReader fr = new FileReader(args[0]);
        xr.parse(new InputSource(fr));
    }
```



Reader 必须被包装在一个 InputSource 对象里才能被浏览。这又是一个适配器模式的应用。

到现在为止，完整地给出所有的源代码，如代码清单 9 所示。

代码清单 9：MySAXApp 类的完整源代码

```
package com.javapatterns.xmlproperties;
import java.io.FileReader;
import org.xml.sax.XMLReader;
import org.xml.sax.InputSource;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;
public class MySAXApp extends DefaultHandler
{
    public MySAXApp() { super(); }
    public static void main(String[] args)
        throws Exception
    {
        XMLReader xr =
            XMLReaderFactory.createXMLReader();
        MySAXApp handler = new MySAXApp();
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);
        FileReader fr = new FileReader(args[0]);
        xr.parse(new InputSource(fr));
    }
}
```

现在可以试着编译此程序并运行它。当然，除非 XML 文件里有不合规范的 XML 标识，否则运行没有任何可以看见的结果。这是因为虽然登记了 MySAXApp 类为事件处理器，但是 MySAXApp 并没有处理任何事件的方法。

事件的处理

当往 MySAXApp 类上加上适当的方法处理 XML 浏览的事件时，事情才变得有意义了。客户端需要实现的最重要的事件便是文件的开始和结束事件。为了处理这两个重要事件，需要在 MySAXApp 类里实现 startDocument 和 endDocument 方法，如代码清单 10 所示。

代码清单 10：怎样在 MySAXApp 类里处理 startDocument 和 endDocument 事件

```
public void startDocument()
{
    System.out.println("startDocument()");
}
public void endDocument()
{
    System.out.println("endDocument()");
}
```

读者可以看到，startDocument 和 endDocument 事件处理方法没有参量。当 SAX 的 driver 发现所浏览的文件的开头和结尾时，driver 就会一次性激发 startDocument()和 endDocument()方法。应当指出的是，即便在浏览过程中有错误发生，在浏览到文件末尾时，endDocument()事件仍会发生。

在前面的例子里，仅仅向命令窗打印出一行字。在实际的应用中可能需要做些很复杂的事情，比如向数据库写记录；刷新图形界面，写上新的数据；或在内存里建立一个数据结构（如对象树结构）等。

SAX 的 driver 也同样会以同样的方法通知 MySAXApp 类一个 XML 元素的开始和结束。与文件的开始和结束不同的是，处理元素的开始和结束事件的方法必须有相应的参数传入。这两个方法在一个元素每次开始和结束时都会打印一行信息出来。URI 是 XML 的名域（Name Space），而 qName 则是 XML 1.0 的用名，在不提供域名 URI 的情况下，qName 一定是 XML1.0 的原始用名。

本节这个简短的介绍不去接触 attributes 的结构。如果读者感兴趣的话，可以进一步阅读有关著作。

最后，SAX2 遇到普通的 character 数据时会通过 character()方法报告给用户端，而 character()方法的使用办法如代码清单 11 所示。

代码清单 11：怎样在 MySAXApp 类里处理 character 事件

```
public void character(char ch[],
    int start, int length)
{
    System.out.println("character()");
    for (int i = start; i < start + length; i++)
    {
        System.out.pritnln(ch[i]);
    }
}
```

MySAXApp 类完整的源代码如代码清单 12 所示。

代码清单 12：MySAXApp 类的完整源代码

```
package com.javapatterns.xmlproperties;
import java.io.FileReader;
import org.xml.sax.XMLReader;
```

```
import org.xml.sax.InputSource;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;
public class MySAXApp extends DefaultHandler
{
    public MySAXApp() { super(); }
    public static void main(String[] args)
        throws Exception
    {
        XMLReader xr =
            XMLReaderFactory.createXMLReader();
        MySAXApp handler = new MySAXApp();
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);

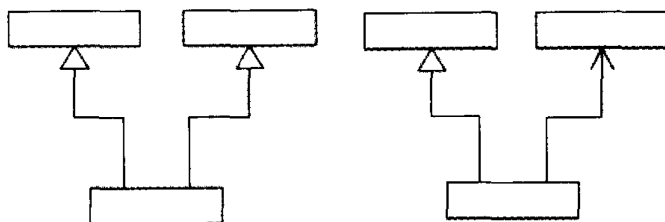
        FileReader fr = new FileReader(args[0]);
        xr.parse(new InputSource(fr));
    }
    public void startDocument()
    {
        System.out.println("startDocument()");
    }
    public void endDocument()
    {
        System.out.println("endDocument()");
    }
    public void character(char ch[], int start,
        int length)
    {
        System.out.println("character()");
        for (int i = start; i < start + length; i++)
        {
            System.out.pritnln(ch[i]);
        }
    }
}
```

24.3 开始 XMLProperties 项目

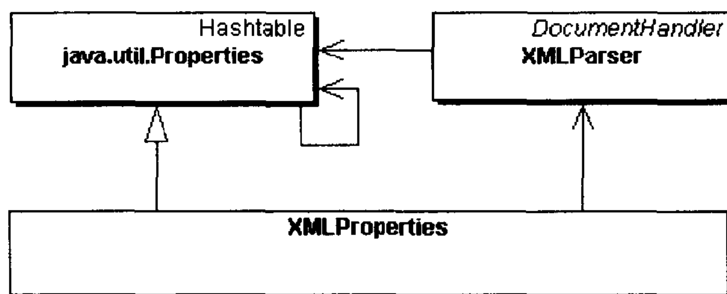
适配器模式的应用

本项目的目的是使用适配器模式将 DocumentHandler 的接口适配成为 java.util.Properties 的接口。适配器模式的简略类图如下图所示，左边是类的适配器模式，右边是

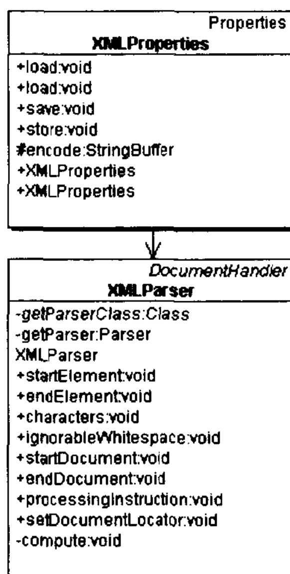
对象的适配器模式。



换言之，如果采取对象的适配器模式的话，XMLProperties 是一个继承自 java.util.Properties 的对象，并委派一个 DocumentHandler 类型的对象，XMLProperties 系统的类图如下图所示。



这就是说，本项目需要两个新类——XMLProperties 类和 XMLParser 类，前者的类型是 java.util.Properties，后者的类型是 DocumentHandler。为了更清晰地看到这两个类的细节，特提供它们的类图如下图所示。



源代码

适配器类 XMLProperties 类的源代码，如代码清单 13 所示。

代码清单 13：XMLProperties 类的完整源代码

```
package com.javapatterns.xmlproperties;
import org.xml.sax.DocumentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.Locator;
import org.xml.sax.AttributeList;
import org.xml.sax.Parser;
import org.xml.sax.InputSource;
import java.io.IOException;
import java.io.OutputStream;
import java.io.InputStream;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.File;
import java.io.PrintWriter;
import java.util.Properties;
import java.util.Enumeration;
/**
 * 此类是一个 Properties 数据和 XML 数据之间的转换器
 *
 * 这里的 XML 格式如下:
 *
 *    <properties
 *        <key name="My_key1"My_Value1</key
 *        <key name="My_key2"My_Value2</key
 *    </properties
 *
 */
public class XMLProperties extends Properties
{
    XMLParser p = null;
    /**
     * 从一个输入流读入 XML
     */
    public synchronized void load(InputStream in)
        throws IOException
    {
        try
        {
            p = new XMLParser(in, this);
        }
        catch (SAXException e)
        {
            throw new IOException(e.getMessage());
        }
    }
```

```
/**
 * 将 XML 文件读入
 */
public synchronized void load(File file)
    throws SAXException,  IOException
{
    InputStream in = new BufferedInputStream(
        new FileInputStream(file));
    XMLParser p = null;
    try
    {
        p = new XMLParser(in, this);
    }
    catch (SAXException e)
    {
        System.out.println(e);
        throw e;
    }
}
/**
 * 调用 store(OutputStream out, String header) 方法
 */
public synchronized void save(OutputStream out,
    String header)
{
    try
    {
        store(out,  header);
    }
    catch (IOException ex)
    {
        ex.printStackTrace();
    }
}
/**
 * 将此 property 列表写入到输出流里
 */
public synchronized void store(OutputStream out,
    String header) throws IOException
{
    PrintWriter wout = new PrintWriter(out);
    wout.println("<?xml version='1.0'?>");
    if (header != null)
    {
        wout.println("<!--" + header + "-->");
    }
```

```
        wout.print("<properties>");
        for (Enumeration e = keys() ;
             e.hasMoreElements() ;)
        {
             String key = (String)e.nextElement();
             String val = (String)get(key);
             wout.print("\n <key name=\"" + key + "\">");
             wout.print(encode(val));
             wout.print("</key>");
        }
        wout.print("\n</properties>");
        wout.flush();
    }
    protected StringBuffer encode(String string)
    {
        int len = string.length();
        StringBuffer buffer = new StringBuffer(len);
        char c;
        for (int i = 0 ; i <len ; i++)
        {
             switch (c = string.charAt(i))
             {
             case '&':
                  buffer.append("&amp;");
                  break;
             case '><':
                  buffer.append("<");
                  break;
             case '':
                  buffer.append(">");
                  break;
             default:
                  buffer.append(c);
             }
        }
        return buffer;
    }
    /**
     * 创建一个没有默认内容的空的 property
     */
    public XMLProperties()
    {
        super();
    }
    /**
     * 创建一个空的 property，并以传入的 property 为默认值
```



```
    */
    public XMLProperties(Properties defaults)
    {
        super(defaults);
    }
}
```

下面是 XMLParser 类的完整源代码，如代码清单 14 所示。

代码清单 14：XMLParser 的完整源代码

```
package com.javapatterns.xmlproperties;
import org.xml.sax.DocumentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.Locator;
import org.xml.sax.AttributeList;
import org.xml.sax.Parser;
import org.xml.sax.InputSource;
import java.io.IOException;
import java.io.OutputStream;
import java.io.InputStream;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.File;
import java.io.PrintWriter;
import java.util.Properties;
import java.util.Enumeration;
class XMLParser implements DocumentHandler
{
    private static final int IN_NOTHING   = 0;
    private static final int IN_DOCUMENT = 1;
    private static final int IN_KEY        = 2;
    private int state = IN_NOTHING;
    private String        key;
    private StringBuffer value;
    private Parser        parser;
    private Class parser_class = null;
    private Properties xmlprop = null;
    public static final String PARSER_P =
        "org.apache.xerces.parsers.SAXParser";
    private Class getParserClass()
        throws ClassNotFoundException
    {
        if (parser_class == null)
            parser_class = Class.forName(PARSER_P);
        return parser_class;
    }
    private Parser getParser()
```

```
{
    try
    {
        return (Parser)
            getParserClass().newInstance();
    }
    catch (Exception ex)
    {
        throw new RuntimeException(
            "Unable to intantiate : "+ PARSER_P);
    }
}
XMLParser(InputStream in, XMLProperties xmlprop)
    throws IOException， SAXException
{
    this.xmlprop = xmlprop;
    state = IN_NOTHING;
    value = new StringBuffer();
    try
    {
        parser = getParser();
        parser.setDocumentHandler(this);
        parser.parse(new InputSource(in));
    }
    catch (Exception e)
    {
        e.printStackTrace();
        throw new SAXException(
            "can't create parser ");
    }
}
public void startElement(String name,
    AttributeList atts) throws SAXException
{
    if (state == IN_NOTHING)
    {
        if (name.equals("properties"))
        {
            state = IN_DOCUMENT;
        }
        else
        {
            throw new SAXException(
                "attempt to find root properties");
        }
    }
```

```
        else if (state == IN_DOCUMENT)
        {
            if (name.equals("key"))
            {
                state = IN_KEY;
                key = atts.getValue("name");
                if (key == null)
                {
                    throw new SAXException(
                        "no name for key "+atts);
                }
            }
            else
            {
                throw new SAXException(
                    "attempt to find keys");
            }
        }
        else
        {
            throw new SAXException(
                "invalid element "+name);
        }
    }
    public void endElement(String name)
        throws SAXException
    {
        if (state == IN_KEY)
        {
            xmlprop.setProperty(key, value.toString());
            System.out.print("<key name=\""+key+"\">");
            System.out.println(
                value.toString()+"</key>\n");
            state = IN_DOCUMENT;
            name = null;
            value = new StringBuffer();
        }
        else if (state == IN_DOCUMENT)
        {
            state = IN_NOTHING;
        }
    }
    public void characters(char[] ch, int   start,
        int length) throws SAXException
    {
        if (state == IN_KEY)
```

```
        {
            compute(ch,   start,   length);
        }
    }
    public void ignorableWhitespace(char[] ch,
        int  start, int length) throws SAXException
    {
        // nothing to do
    }
    public void startDocument() throws SAXException
    {
        // nothing to do
    }
    public void endDocument() throws SAXException
    {
        // nothing to do
    }
    public void processingInstruction(String target,
        String data) throws SAXException
    {
        // nothing to do
    }
    public void setDocumentLocator(Locator locator)
    {
        // nothing to do
    }
    private void compute(char[] ch, int  start,
        int length)
    {
        int st = start;
        int len = length-1;
        while (st <length
            && ((ch[st] == '\n') || (ch[st] == '\t')
                || (ch[st] == ' ')
                || (ch[st] == '\r'))) {
              st++;
        }
        while (len > 0
            && ((ch[len] == '\n')
                || (ch[len] == '\t')
                || (ch[len] == ' ')
                || (ch[len] == '\r'))) {
              len--;
        }
        while (st <= len)
        {
```

```
                value.append(ch[st]);
                st++;
            }
        }
    }
```

示意性的客户端类 TestXML 的源代码，如代码清单 15 所示。

代码清单 15：TestXML 的源代码

```
package com.javapatterns.xmlproperties;
import org.xml.sax.SAXException;
import java.io.IOException;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.util.Enumeration;
public class TestXML
{
    public static void main(String[] args)
        throws SAXException ,  IOException
    {
        XMLProperties xml = new XMLProperties();
        xml.load(new File(
            "C:/javapatterns/mybook/
            xmlproperties/xml2.xml"));
        Enumeration keyEnum = xml.keys();
        Enumeration valueEnum = xml.elements();
        while (keyEnum.hasMoreElements())
        {
             String key = (String) keyEnum.nextElement();
             String value = (String)
                 valueEnum.nextElement();
            System.out.println(key + "=" + value);
        }
        OutputStream out = new
            FileOutputStream(new File("c:/out.xml"));
        xml.store(out,  "test");
    }
}
```

程序的运行

Myxml.xml 的源代码如代码清单 16 所示。

代码清单 16：myxml.xml 的源代码

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE properties SYSTEM "xmlproperties.dtd">
<!-- XML Wrapper for Properties -->
<properties>
<propertylist name="mylist">
<property name="name">
    webendshere
</property>
<property name="website">
    www.webendshere.com
</property>
<property name="email">
    webmaster@webendshere.com
</property>
<property name="techsupport">
    support@webendshere.com
</property>
</propertylist>
</properties>
```

Xmlproperties.dtd 的源代码如代码清单 17 所示。

代码清单 17：xmlproperties.dtd 的源代码

```
<?xml version='1.0' encoding='UTF-8'?>
  <!ELEMENT properties (#PCDATA|property)*>
  <!ELEMENT key (#PCDATA)>
  <!ATTLIST key name CDATA #IMPLIED>
```

要运行这个应用程序，可以使用下面的命令，如代码清单 18 所示。

代码清单 18：运行程序的 Java 指令

```
java com/javapatterns/xmlproperties/TestXML myxml.xml
```

运行的结果如代码清单 19 所示。

代码清单 19：程序的运行结果

```
myxml.xml
Properties loading starts.
Properties loaded
-- listing properties --
mylist.techsupport=support@webendshere.com
mylist.name=webendshere
mylist.website=www.webendshere.com
mylist.email=webmaster@webendshere.com
```

本章的 XMLProperties 的源代码是基于 org.w3c.util.XMLProperties 源代码并经过修改的。

第 25 章　合成（Composite）模式

合成（Composite）模型模式属于对象的结构模式[GOF95]，有时又叫做部分－整体（Part-Whole）模式。合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式可以使客户端将单纯元素与复合元素同等看待。

25.1　对象的树结构

树结构在过程性的编程语言中曾经发挥了巨大的作用。在面向对象的语言中，树结构也同样威力巨大。一个基于继承的类型的等级结构便是一个树结构；一个基于合成的对象结构也是一个树结构。本章将要介绍的合成模式也是一个处理对象的树结构的模式。

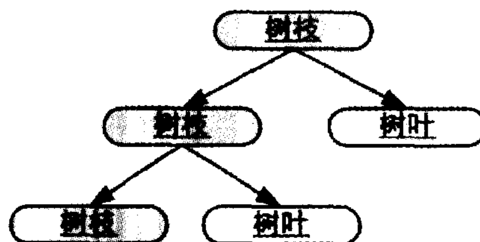
本章只考虑连通并且有方向的树结构。

有向树结构的种类

有向树结构又可以分为三种：从上向下、从下向上和双向的。在三种有向树图中，树的节点和它们的相互关系都是一样的，但是连接它们的关系的方向却很不一样。

由上向下的树图

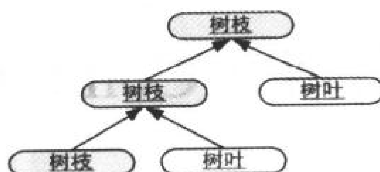
在由上向下的树图中，每一个树枝节点都有箭头指向它的所有的子节点，从而一个客户端可以要求一个树枝节点给出所有的子节点，而一个节点却并不知道它的父节点。在这样的树结构上，信息可以按照箭头所指的方向自上向下传播。由上向下的树图如下图所示。



由下向上的树图

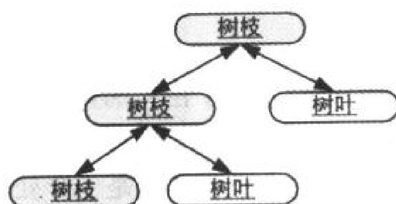
在一个由下向上的树图中，每一个节点都有箭头指向它的父节点，但是一个父节点却不知道其子节点。信息可以按照箭头所指的方向自下向上传播。由下向上的树图如下图所

示。



双向的树图

在一个双向的树图中，每一个节点都同时知道它的父节点和所有的子节点。在这样的树结构上，信息可以按照箭头所指的方向向两个方向传播。双向树图如下图所示。



树图中的两种节点

一个树结构由两种节点组成：树枝节点和树叶节点。树枝节点可以有子节点，而一个树叶节点不可以有子节点。



一个树枝节点可以不带有任何叶子，但是它因为有带有叶子的能力，因此仍然是树枝节点，而不会成为叶子节点。一个树叶节点则永远不可能带有子节点。

在信息系统里面，树枝节点所代表的构件常常用做树叶节点所代表的构件的容器。

根节点

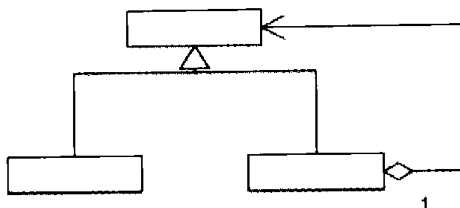
一个树结构中总有至少一个节点是特殊的节点，称做根节点。一个根节点没有父节点，因为它是树结构的根。一般讨论的树都是只有一个根节点的树。

一个树的根节点一般是树枝节点，如果根节点是树叶节点的话，这个树就变成了只有这一个节点的树。

树结构的类图

可以使用类图描述一个树结构的静态结构。下图所示的是合成模式的简略类图，同时也是一个典型的树结构的类图。





可以看出，最上方出现的是一个抽象的节点，左下方的是一个树叶节点，而右上方出现的是一个树枝节点，它含有其他的节点。

25.2　介绍合成模式

从道士的故事谈起

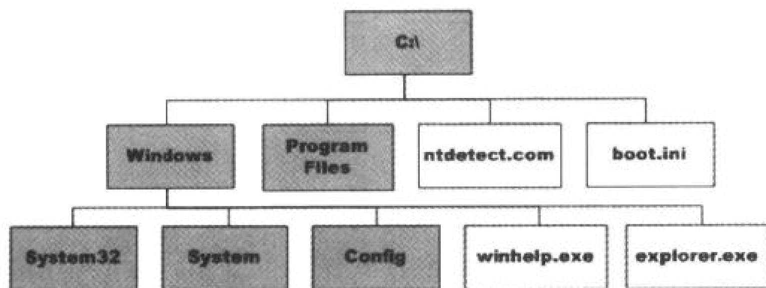
这是小的时候我奶奶讲的故事：从前有个山，从前有个庙，从前有个道士讲故事。讲的啥故事？从前有个山，从前有个庙，从前有个道士讲故事。讲的啥故事？从前有个山……奶奶的故事要循环多少次，根据你多长时间睡着而定。在故事里有山、有庙、有道士、有故事。因此，故事的角色有两种：一种里面没有其他角色；另一种内部有其他角色。

合成模式

合成模式把部分和整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由它们复合而成的合成对象同等看待。

文件系统

再比如，一个文件系统就是一个典型的合成模式系统。下图所示就是常见的 PC 文件系统的一部分。



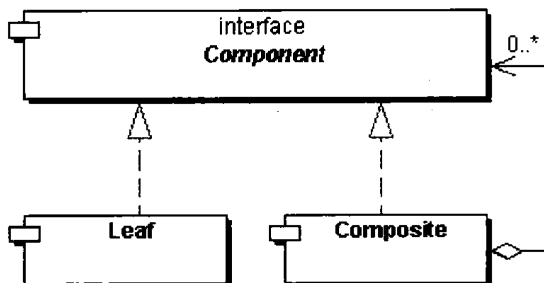
从上图可以看出，文件系统是一个树结构，树上长有节点。树的节点有两种，一种是树枝节点，即目录，有内部树结构，在图中涂有颜色；另一种是文件，即树叶节点，没有

内部树结构。这里暂时不考虑快捷方法（shortcut）这种特殊的节点，留待讨论代理模式时一同讨论。

显然，可以把目录和文件当做同一种对象同等对待和处理，这也就是合成模式的应用。

25.3 安全式和透明式的合成模式

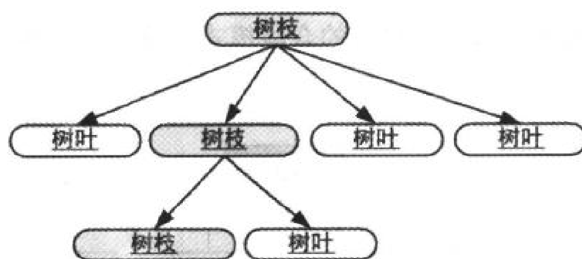
下图所示的类图略去了各个角色的细节，没有给出它们的各种方法。



可以看出上面的类图结构涉及到三个角色：

- 抽象构件（Component）角色：这是一个抽象角色，它给参加组合的对象规定一个接口。这个角色给出共有的接口及其默认行为。
- 树叶构件（Leaf）角色：代表参加组合的树叶对象。一个树叶没有下级的子对象。定义出参加组合的原始对象的行为。
- 树枝构件（Composite）角色：代表参加组合的有子对象的对象，并给出树枝构件对象的行为。

可以看出，Composite 类型的对象含有其他的 Component 类型的对象。换言之，Composite 类型的对象可以含有其他的树枝（Composite）类型或树叶（Leaf）类型的对象。一个典型的 Composite 对象图如下图所示。



合成模式的实现根据所实现接口的区别分为两种形式，分别称为安全式和透明式。虽然这是模式的实现问题，但是由于它影响到模式结构的细节，因此本书将它移到模式的结构之前讲解。

合成模式可以不提供父对象的管理方法，但是合成模式必须在合适的地方提供子对象的管理方法。在什么地方声明子对象的管理方法，诸如 add()、remove() 以及 getChild()等

就变成了一个值得仔细思考的问题，可以讨论的选择有如下两个。

透明方式

作为第一种选择，在 Component 里面声明所有的用来管理子类对象的方法，包括 add()、remove()，以及 getChild()方法。这样做的好处是所有的构件类都有相同的接口。在客户端看来，树叶类对象与合成类对象的区别起码在接口层次上消失了，客户端可以同等地对待所有的对象。这就是透明形式的合成模式。

这个选择的缺点是不够安全，因为树叶类对象和合成类对象在本质上是有区别的。树叶类对象不可能有下一个层次的对象，因此 add()、remove() 以及 getChild()方法没有意义，但是在编译时期不会出错，而只会在运行时期才会出错。

安全方式

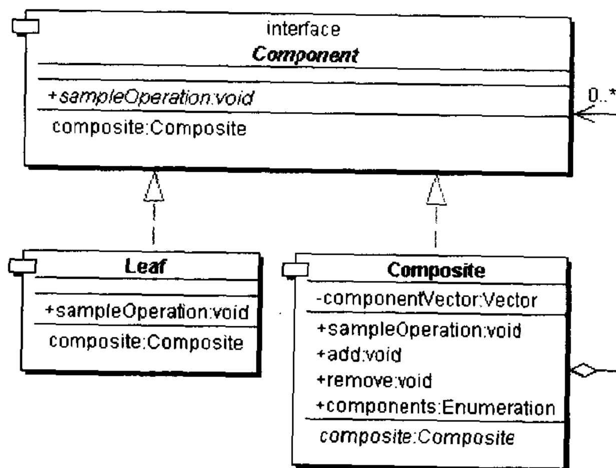
第二种选择是在 Composite 类里面声明所有的用来管理子类对象的方法。这样的做法是安全的做法，因为树叶类型的对象根本就没有管理子类对象的方法，因此，如果客户端对树叶类对象使用这些方法时，程序会在编译时期出错。编译通不过，就不会出现运行时期错误。

这个选择的缺点是不够透明，因为树叶类和合成类将具有不同的接口。

这两个形式各有优缺点，需要根据软件的具体情况做出取舍决定。

25.4　安全式的合成模式的结构

安全式的合成模式要求管理聚集的方法只出现在树枝构件类中，而不出现在树叶构件类中。安全式的合成模式的类图如下图所示。



这种形式涉及到三个角色：

- 抽象构件（Component）角色：这是一个抽象角色，它给参加组合的对象定义出

公共的接口及其默认行为，可以用来管理所有的子对象。合成对象通常把它所包含的子对象当做类型为 Component 的对象。在安全式的合成模式里，构件角色并不定义出管理子对象的方法，这一定义由树枝构件对象给出。

- 树叶构件（Leaf）角色：树叶对象是没有下级子对象的对象，定义出参加组合的原始对象的行为。
- 树枝构件（Composite）角色：代表参加组合的有下级子对象的对象。树枝构件类给出所有的管理子对象的方法，如 add()、remove() 以及 components()的声明。

抽象构件角色由一个 Java 接口实现，它给出两个对所有的子类均有用的方法：getComposite()和 sampleOperation()，如代码清单 1 所示。

代码清单 1：抽象构件角色 Component 的接口代码

```
public interface Component
{
    /**
     *  返还自己的实例
     */
     Composite getComposite();
    /**
     *  某个商业方法
     */
     void sampleOperation();
}
```

树枝构件角色由一个具体 Java 类实现。这个具体类不仅仅给出抽象构件角色所声明的两个方法：getComposite()和 sampleOperation()，而且声明并实现了管理聚集用的 add()、remove() 以及 components()等方法。其中 components()方法就相当于 getChild()方法，如代码清单 2 所示。

代码清单 2：树枝构件（Composite）类的源代码

```
import java.util.Vector;
import java.util.Enumeration;
public class Composite implements Component
{
    private Vector componentVector = new java.util.Vector();
    /**
     *  返还自己的实例
     */
    public Composite getComposite()
    {
        return this;
    }
    /**
     *  某个商业方法
     */
    public void sampleOperation()
```

```
    {
        Enumeration enumeration = components();
        while (enumeration.hasMoreElements())
        {
            ((Component)enumeration.nextElement()).sampleOperation();
        }
    }
    /**
     *  聚集管理方法，增加一个子构件对象
     */
    public void add(Component component)
    {
        componentVector.addElement(component);
    }
    /**
     *  聚集管理方法，删除一个子构件对象
     */
    public void remove(Component component)
    {
        componentVector.removeElement(component);
    }
    /**
     *  聚集管理方法，返还聚集的 Enumeration 对象
     */
    public Enumeration components()
    {
        return componentVector.elements();
    }
}
```

树叶构件角色则仅仅实现了抽象构件角色所要求的两个方法：sampleOperation()和getComposite()，如代码清单 3 所示。

代码清单 3：树叶构件（Leaf）类的源代码

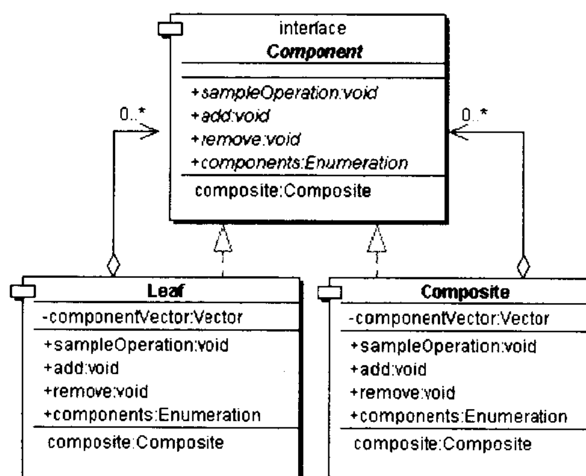
```
import java.util.Enumeration;
public class Leaf implements Component
{
    /**
     *  某个商业方法
     */
    public void sampleOperation()
    {
        // Write your code here
    }
    /**
     *  返还自己的实例
     */
```

```
    public Composite getComposite()
    {
        // Write your code here
        return null;
    }
}
```

可以看出，树枝构件类给出了 add()、remove() 以及 components()等方法的声明和实现，而树叶构件类则没有给出这些方法的声明或实现。由于这个特点，客户端应用程序不可能错误地调用树叶构件的聚集方法，因为树叶构件没有这些方法，调用会导致编译出错。

25.5 透明式的合成模式的结构

与安全式的合成模式不同的是，透明式的合成模式要求所有的具体构件类，不论树枝构件还是树叶构件，均符合一个固定的接口。透明式的合成模式的示意性类图如下图所示。



这种形式涉及到三个角色：

- 抽象构件（Component）角色：这是一个抽象角色，它给参加组合的对象规定一个接口，规范共有的接口及默认行为。这个接口可以用来管理所有的子对象，要提供一个接口以规范取得和管理下层组件的接口，包括 add()、remove() 以及 getChild()之类的方法。
- 树叶构件（Leaf）角色：代表参加组合的树叶对象，定义出参加组合的原始对象的行为。树叶类会给出 add()、remove() 以及 getChild()之类的用来管理子类对象的方法的平庸实现。
- 树枝构件（Composite）角色：代表参加组合的有子对象的对象，定义出这样的对象的行为。

下面给出的是抽象构件角色的源代码，如代码清单 4 所示。可以看出，这是一个 Java 接口，它声明了用于管理聚集的相关方法：add()、remove()、components() 以及 getComposite()

等。

代码清单 4：抽象构件角色 Component 接口的源代码

```
import java.util.Enumeration;
public interface Component
{
    /**
     *  某个商业方法
     */
     void sampleOperation();
    /**
     *  返还自己的实例
     */
     Composite getComposite();
    /**
     *  聚集管理方法，增加一个子构件对象
     */
     void add(Component component);
    /**
     *  聚集管理方法，删除一个子构件对象
     */
     void remove(Component component);
    /**
     *  聚集管理方法，返还聚集的 Enumeration 对象
     */
     Enumeration components();
}
```

树枝构件角色实现了抽象构件角色所规定的所有接口。特别指出的是，树枝构件角色给出了管理聚集的所有方法，包括 add()、remove() 以及 components()等方法的非平庸实现，如代码清单 5 所示。

代码清单 5：树枝构件（Composite）类的源代码

```
import java.util.Vector;
import java.util.Enumeration;
public class Composite implements Component
{
     private Vector componentVector = new Vector();
    /**
     *  返还自己的实例
     */
     public Composite getComposite()
     {
          return this;
     }
    /**
```

```
     * 某个商业方法
     */
    public void sampleOperation()
    {
        Enumeration enumeration = components();
        while (enumeration.hasMoreElements())
        {
            ((Component)enumeration.nextElement()).sampleOperation();
        }
    }
    /**
     * 聚集管理方法，增加一个子构件对象
     */
    public void add(Component component)
    {
        componentVector.addElement(component);
    }
    /**
     * 聚集管理方法，删除一个子构件对象
     */
    public void remove(Component component)
    {
        componentVector.removeElement(component);
    }
    /**
     * 聚集管理方法，返还聚集的 Enumeration 对象
     */
    public Enumeration components()
    {
        return componentVector.elements();
    }
}
```

树叶构件角色也同样实现了抽象构件角色所声明的各个方法，包括管理聚集的各个方法。但是，树叶构件因为没有子节点，因此并没有聚集可以管理，所以它所给出的管理聚集用的各个方法，包括 add()、remove() 以及 components()等方法均是平庸的，如代码清单 6 所示。

代码清单 6：树叶构件（Leaf）类的源代码

```
import java.util.Enumeration;
public class Leaf implements Component
{
    /**
     * 某个商业方法
     */
    public void sampleOperation()
```

```
    {
        // Write your code here
    }
   /**
    *  聚集管理方法，增加一个子构件对象
    */
    public void add(Component component)
    {
    }
   /**
    *  聚集管理方法，删除一个子构件对象
    */
    public void remove(Component component)
    {
    }
   /**
    *  返还自己的实例
    */
    public Composite getComposite()
    {
        return null;
    }
   /**
    *  聚集管理方法，返还聚集的 Enumeration 对象
    */
    public Enumeration components()
    {
        return null;
    }
}
```

可以看出，树叶类给出了 add()、remove() 以及 components()等方法的平庸实现，因为这些方法其实并不适用于树叶类。

25.6　合成模式的实现

实现合成模式时，有几个可以考虑的问题：

（1）明显地给出父对象的引用。在子对象里面给出父对象的引用，这样可以很容易地遍历所有的父对象，管理合成结构。有了这个引用，可以方便地应用责任链模式。

定义出这个父对象引用的恰当地方就是 Composite 角色。树叶对象和合成对象都从这个父对象继承对父对象的引用。

在一个子对象被加到合成对象里面的时候，需要修改父对象的记录。当把子对象从合成对象里删除时，也需要改变父对象的记录。因此，需要在所有的构件对象里实现 add()

和 remove()方法。调用 add()方法把一个子对象加到合成对象上；调用 remove()方法把一个子对象从合成对象中删除掉。

一般而言，一个对象持有对所有的子对象的引用，而没有对父对象的引用。当子对象持有对父对象的引用时，树结构就成为由下向上的树结构。当一个对象持有对所有的子对象的引用时，树结构就成为由上向下的树结构。

当然，如果一个对象既持有对所有子对象的引用，又持有对其父对象的引用时，树结构就是双向的树结构。

在本章所给出的所有讨论里，都采取了由上向下的树结构。

（2）在通常的系统里，可以使用享元模式实现构件的共享，但是由于合成模式的对象经常要有对父类的引用，因此共享不容易实现。

（3）抽象构件类（Component）应当多"重"才算好。

一种思路是抽象构件类应当尽量"重"。选用这种形式的合成模式的目的便是使客户端不知道它所使用的是哪一个特殊的树叶构件或树枝构件。这意味着所有的具体构件类都有相同的接口，定义出这个接口的当然应当是抽象构件（Component）。

但是，如果抽象构件同时定义出树叶构件和树枝构件的接口，这便意味着有一些方法适用于树叶构件而不适用于树枝构件；而另外有一些方法适用于树枝构件而不适用于树叶构件。比如，树枝构件有子构件，而树叶构件没有子构件。因为一个叶子对象没有子对象，在叶子对象里下面的方法就需要返还 null：

```
public Component getChild(int  index)
```

另一种思路是使 Component 类较"轻"。因为管理子对象的这部分接口不适用于树叶部件，所以避免把合成部件特有的接口移到抽象部件中去。

这实际上是前面讨论过的安全式和透明式的合成模式，只是换了一种表达方式。

（4）有时候系统需要遍历过一个树枝构件的子构件很多次，这时候就可以把遍历子构件的结果暂时存放在父构件里面，作为缓存。

（5）使用什么数据类型来存储子对象。在给出的示意性定义代码里面，使用了 Vector 来存储合成对象所含有的子对象。但是，在实际系统里不一定要使用 Vector，也可以使用数组的其他聚集。

由于 Java 语言的垃圾收集器可以自动地收集走不被引用的对象，这使得合成模式所涉及的对象可以在不被使用时自动被清除掉。在其他的没有垃圾收集器的语言里，还需要考虑在什么地方装置一个对象和子对象的删除方法的问题。

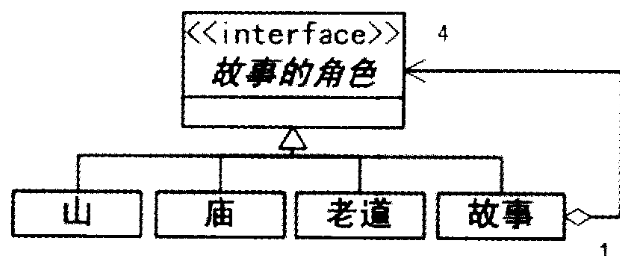
（6）Composite 向子类的委派。客户端不应当直接调用树叶类，应当由其父类向树叶类进行委派。这样可以增加代码的复用性。

25.7 道士的故事

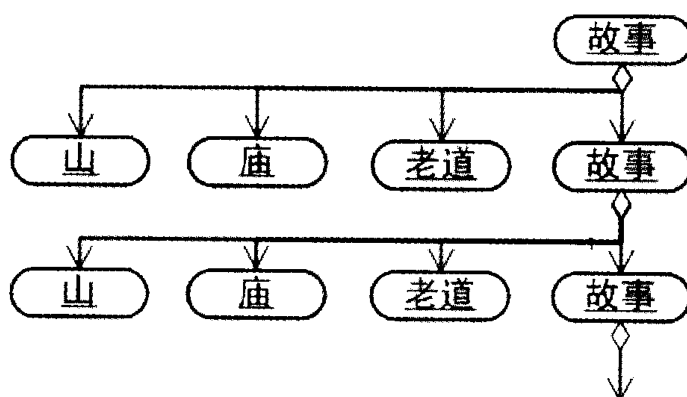
现在回到我奶奶讲的故事。在这个故事里面，有两种角色：一种是没有内部角色的角色，例如山、庙、道士；另一种是有内部角色的角色，在这里就是故事。故事里面又有山、

庙、道士、故事等。

道士的故事的静态结构如下图所示。



从“道士的故事”的对象图可以清楚地看出，故事里面有山对象、庙对象、道士对象和故事对象，如此循环不已。其模拟系统如下图所示。



这种形式涉及到三个角色：

- 抽象构件（Component）角色：由故事角色接口扮演。这是一个抽象角色，给所有的角色对象规定一个接口，可以用来管理所有的诸如山、庙、道士和故事对象。
- 树叶构件（Leaf）角色：由山、庙和道士扮演。树叶对象没有下级的子对象。
- 树枝构件（Composite）角色：由故事角色扮演。

25.8　一个绘图的例子

本节给出一个绘图软件的例子，以说明合成模式的应用。一个绘图系统给出各种工具用来描绘由线、长方形和圆形等基本图形组成的图形。显然，一个复合的图形既然是由这些基本图形组成的，那么合成模式就是一个合适的设计模式。

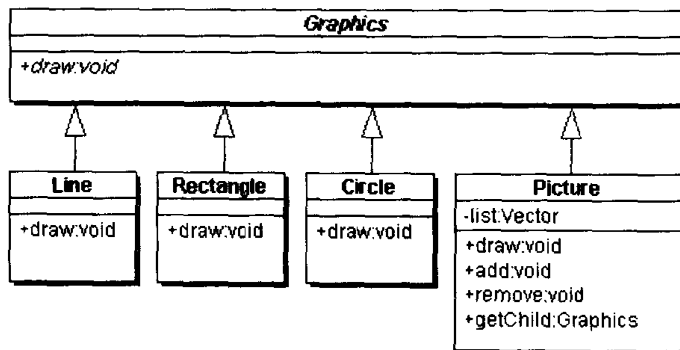
可以想像，设计应当包括 Line、Rectangle 和 Circle 等对象，而且每一个对象都应当配备有一个 draw()方法，在调用时，会划出对象所代表的图形。

同时，由于一个复杂的图形是由基本图形组合而成的，因此，一个合成的图形应当有一个列表，存储对所有的基本图形对象的引用。复合图形的 draw()方法在调用时，应当逐一调用所有列表上的基本图形对象的 draw()方法。

同样，由于合成模式的两种形式，本节同样给出两种实现。

应用安全式合成模式

安全形式的合成模式意味着只有树枝构件角色才配备有管理聚集的方法，而树叶构件则没有这些方法。下图所示就是使用安全形式的合成模式给出的设计结构图。



可以看出，客户端可以调用 add()方法加入一个基本图形，或者调用 remove()方法取消一个基本图形，或者 getChild(index)以得到组成复杂图形的第 *i* 个基本图形。在这样做的时候，客户端必须肯定这是一个复合图形而不是一个基本图形，因为基本图形对象并没有这些用来管理聚集的方法。

下面就是抽象构件角色的源代码，如代码清单 7 所示。可以看出，此抽象角色仅仅声明了一个商业方法，而没有用于操作子对象聚集的方法，因此，这里使用的是安全类型的合成模式。

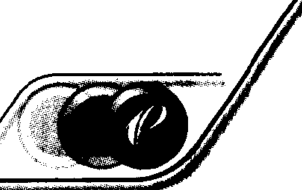
代码清单 7：抽象构件角色的实现

```
package com.javapatterns.composite.drawingsafe;
abstract public class Graphics
{
    public abstract void draw();
}
```

Picture 类是树枝构件角色，它实现了抽象构件角色所要求的方法，而且还额外提供了用于管理子对象聚集的一系列方法，如代码清单 8 所示。

代码清单 8：树枝构件角色的实现

```
package com.javapatterns.composite.drawingsafe;
import java.util.Vector;
public class Picture extends Graphics
{
    private Vector list = new Vector(10);
    public void draw()
    {
        for (int i = 0 ; i < list.size(); i++)
        {
            Graphics g = (Graphics) list.get(i);
```



```
            g.draw();
        }
    }
    /**
     *  聚集管理方法，增加一个子构件对象
     */
    public void add(Graphics g)
    {
        list.add(g);
    }
    /**
     *  聚集管理方法，删除一个子构件对象
     */
    public void remove(Graphics g)
    {
        list.remove(g);
    }
    /**
     *  返还一个子构件对象
     */
    public void getChild(int i)
    {
        return (Graphics) list.get(i);
    }
}
```

Line 是一个树叶构件，它没有任何的子对象，因此也不必提供管理子对象聚集的方法，如代码清单 9 所示。

代码清单 9：Line 作为一个树叶构件的实现

```
package com.javapatterns.composite.drawingsafe;
public class Line extends Graphics
{
    public void draw()
    {
        //write your code here
    }
}
```

下面是树叶构件角色 Rectangle 的源代码，如代码清单 10 所示。

代码清单 10：Rectangle 作为一个树叶构件的实现

```
package com.javapatterns.composite.drawingsafe;
public class Rectangle extends Graphics
{
    public void draw()
    {
```

```
            //write your code here
        }
    }
```

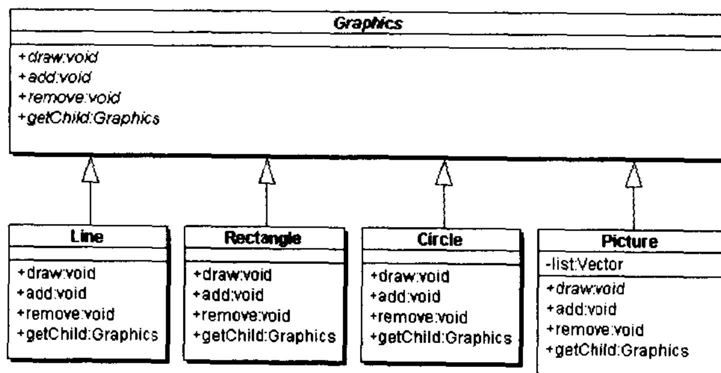
下面是树叶构件角色 Circle 类的源代码，如代码清单 11 所示。

代码清单 11：Circle 作为一个树叶构件的实现

```
    package com.javapatterns.composite.drawingsafe;
    public class Circle extends Graphics
    {
        public void draw()
        {
            //write your code here
        }
    }
```

应用透明式的合成模式

透明形式的合成模式意味着不仅只有树枝构件角色才配备有管理聚集的方法，树叶构件也有这些方法，虽然树叶构件的这些方法是平庸的。下图所示就是使用透明形式的合成模式给出的设计图。



可以看出，与安全形式的模式不同的是，一个基本图形对象和一个复合图形对象一样，同样具有管理聚集的那些方法，虽然基本图形对象所具有的那些方法是平庸的。因此，客户端无需检查调用 add()方法加入一个基本图形，或者调用 remove()方法取消一个基本图形，或者调用 getChild(index)以得到组成复杂图形的第 i 个基本图形。在这样做的时候，客户端必须肯定这是一个复合图形而不是一个基本图形，因为基本图形对象并没有这些用来管理聚集的方法。

下面给出的是抽象构件角色的源代码，如代码清单 12 所示，它定义出所有的具体构建类所必须实现的接口。显然，由于此抽象角色声明了操作子对象聚集的各个方法，因此，这里使用的是透明形式的合成模式。

代码清单 12：抽象构件角色的实现

```
    package com.javapatterns.composite.drawingtransaprent;
    abstract public class Graphics
```

```
{
    public abstract void draw();
   /**
    *  聚集管理方法，增加一个子构件对象
    */
    public abstract void add(Graphics g);
   /**
    *  聚集管理方法，删除一个子构件对象
    */
    public abstract void remove(Graphics g);
   /**
    *  返还一个子构件对象
    */
    public abstract Graphics getChild(int i);
}
```

下面是具体构建子类的源代码，如代码清单 13 所示，它实现了抽象角色所声明出的所有的方法。但是，必须指出，由于这个具体子类是一个树枝构件，因此它给出了管理子对象聚集的相应的各个方法的有意义的实现。

代码清单 13：树枝构件角色的实现

```
package com.javapatterns.composite.drawingtransaprent;
import java.util.Vector;
public class Picture extends Graphics
{
    private Vector list = new Vector(10);
    public void draw()
    {
        for (int i = 0 ; i < list.size(); i++)
        {
            Graphics g = (Graphics) list.get(i);
            g.draw();
        }
    }
   /**
    *  聚集管理方法，增加一个子构件对象
    */
    public void add(Graphics g)
    {
        list.add(g);
    }
   /**
    *  聚集管理方法，删除一个子构件对象
    */
    public void remove(Graphics g)
    {
```

```
            list.remove(g);
        }
        /**
         *  返还一个子构件对象
         */
        public Graphics getChild(int i)
        {
            return (Graphics) list.get(i);
        }
}
```

Line 类是树叶构件角色，但由于它必须实现抽象构件角色所要求的方法，因此，Line 类提供了用于管理子对象聚集的一系列方法。由于这个类是树叶类，根本不会有任何子对象，因此这些聚集管理方法实际上是空的，如代码清单 14 所示。

代码清单 14：Line 作为一个树叶构件角色的实现

```
package com.javapatterns.composite.drawingtransaprent;
public class Line extends Graphics
{
    public void draw()
    {
        //write your code here
    }
    /**
     *  聚集管理方法，增加一个子构件对象
     */
    public void add(Graphics g)
    {
        //do nothing
    }
    /**
     *  聚集管理方法，删除一个子构件对象
     */
    public void remove(Graphics g)
    {
        //do nothing
    }
    /**
     *  返还一个子构件对象
     */
    public Graphics getChild(int i)
    {
        return null;
    }
}
```

Rectangle 和 Circle 与 Line 一样，都是树叶构件角色，如代码清单 15 和代码清单 16

所示。

代码清单 15：Rectangle 作为一个树叶构件角色的实现

```
package com.javapatterns.composite.drawingtransaprent;
public class Rectangle extends Graphics
{
    public void draw()
    {
        //write your code here
    }
   /**
    *  聚集管理方法，增加一个子构件对象
    */
    public void add(Graphics g)
    {
        //do nothing
    }
   /**
    *  聚集管理方法，删除一个子构件对象
    */
    public void remove(Graphics g)
    {
        //do nothing
    }
   /**
    *  返还一个子构件对象
    */
    public Graphics getChild(int i)
    {
        return null;
    }
}
```

代码清单 16：Circle 作为一个树叶构件角色的实现

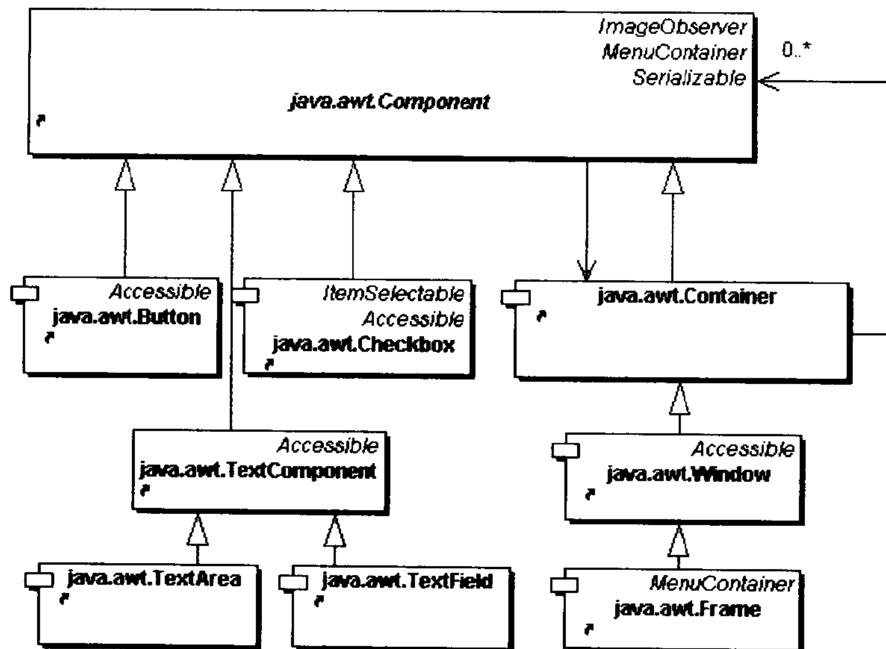
```
package com.javapatterns.composite.drawingtransaprent;
public class Circle extends Graphics
{
    public void draw()
    {
        //write your code here
    }
   /**
    *  聚集管理方法，增加一个子构件对象
    */
    public void add(Graphics g)
    {
        //do nothing
```

```
    }
    /**
     * 聚集管理方法，删除一个子构件对象
     */
    public void remove(Graphics g)
    {
        //do nothing
    }
    /**
     * 返还一个子构件对象
     */
    public Graphics getChild(int i)
    {
        return null;
    }
}
```

25.9 AWT 库中的例子

由于 AWT 和 Swing 的图形界面构件是建立在 AWT 库中的 Container 类和 Component 类上的，从下面的 AWT 合成模式类图可以看出，Button 和 Checkbox 是树叶型的构件，而 Container 则是树枝型的构件。

这意味着在 Container 对象中可以含有其他的 Component 对象，如下图所示。



除了 Button 和 Checkbox 构件之外，还有诸如 TextComponent，以及 TextComponent 的子类，如 TextArea 和 TextField 等。

在 Container 类中，有操作聚集的对应方法，而在 Component 类中则没有这样的方法。这就是说，AWT 中使用的合成模式是安全形式的合成模式。

在 AWT 库中使用了合成模式的例子实际上也就是上面这些了，但是在具体应用 AWT 库时，读者会大量使用合成模式。

在下面的情况下应当考虑使用合成模式：

- 需要描述对象的部分和整体的等级结构。
- 需要客户端忽略掉个体构件和组合构件的区别。客户端必须平等对待所有的构件，包括个体构件和组合构件。

合成模式有优点和缺点。它的优点有：

- 合成模式可以很容易地增加新种类的构件。
- 使用合成模式可以使客户端变得很容易设计，因为客户端不需要知道构件是树叶构件还是树枝构件。

合成模式的缺点有：

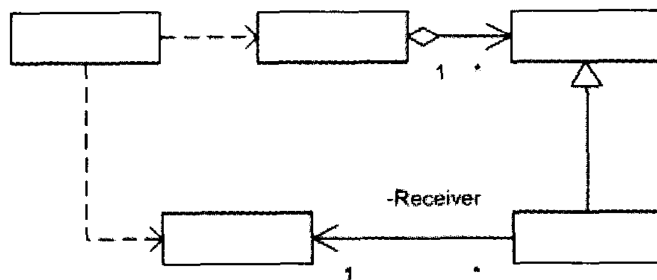
- 使用合成模式后，控制树枝构件的类型就不太容易。
- 用继承的方法来增加新的行为很困难，具体请见下面“合成模式与装饰模式的关系”一节。

25.10　合成模式与其他模式的关系

与合成模式有关系的模式有命令模式、装饰模式、迭代模式及责任链模式和享元模式。

合成模式与命令（Command）模式的关系

合成模式常常可以应用到命令类的合成上，由几个具体命令类合成宏命令类。命令模式的简略类图如下图所示。

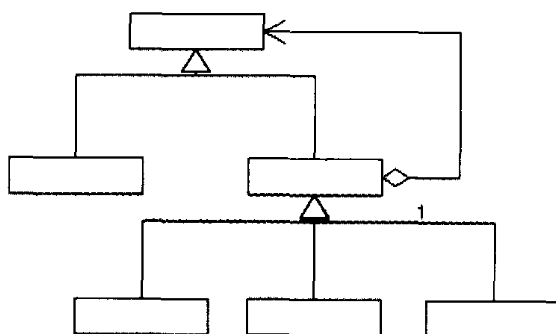


合成模式与装饰（Decorator）模式的关系

用继承关系为装饰模式增加新的行为很困难。如果建立一个抽象构件类的子类，当然可以把新的行为添加到抽象子类中去，但是这样并不能影响到已有的合成类和树叶类的行

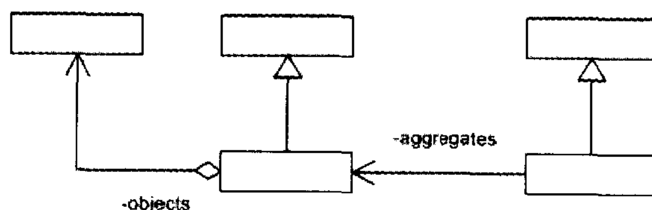
为。为了能将新的行为添加到已有的合成类和树叶类中去，就必须建立符合类和树叶类的子类，这就意味着要修改或复制所有的已有的类。

如果建立合成类的子类，把新行为添加到它里面的话，为了保持多态性，就只好为树叶类也建立子类，并把新行为添加到里面。最后，也必须对抽象构件类这样做。装饰模式的简略类图如下图所示。



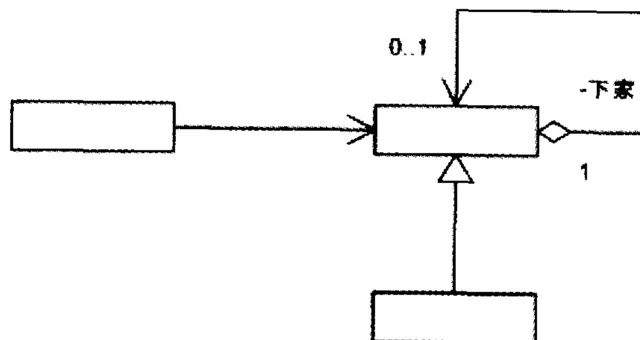
合成模式与迭代（Iterator）模式的关系

合成模式常常用迭代子模式遍历子类对象，迭代模式的简略类图如下图所示。



合成模式与责任链（Chain of Responsibility）模式的关系

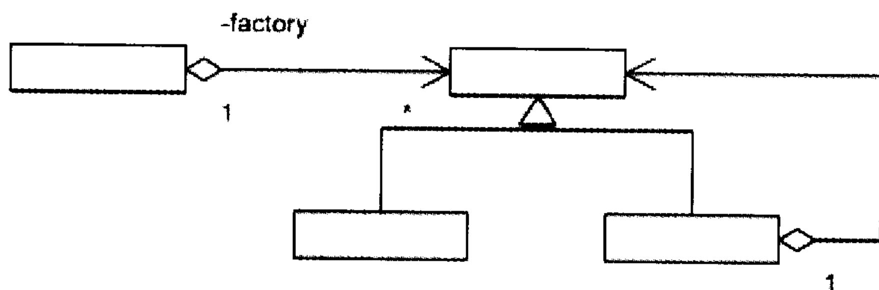
一个责任链模式往往应用到一个树结构上，因此责任链模式会使用到合成模式、迭代模式和装饰模式。责任链模式的类图如下图所示。



合成模式与享元模式（Flyweight Pattern）的关系

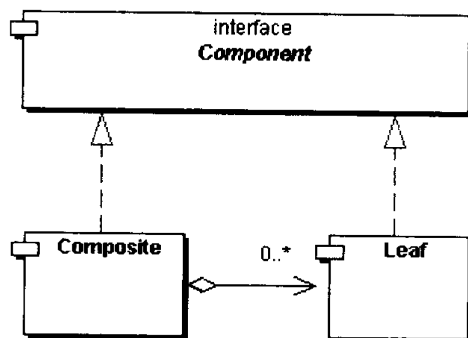
严格地讲，享元模式（Flyweight Pattern）并不是一个单纯的模式，而是一个由数个模式组合而成的复合模式。享元模式的工厂角色是一个工厂方法模式，工厂方法的内部记录了所创建的产品实例，并循环使用这些事例。

复合享元对象是合成模式的应用，抽象享元角色就是复合构件角色，而具体享元角色就是具体构件角色。复合享元是树枝构件，而单纯享元是树叶构件。享元模式的类图如下图所示。



问答题

1. 在模式的定义图中，如果聚合的关系不是从 Composite 到 Component 的，而是从 Composite 到 Leaf 的（如下图所示），会怎么样呢？



2. 请看一看算术运算的逻辑和结构。比如：1+2 就是一个算术运算，它的运算符号是+，两个运算量分别是 1 和 2。算术运算涉及两个运算量和一个运算符号，运算符号可以是+、–、*、/；运算量有两种，一种是纯数字，像 12、34，另一种是一个算术运算式。

换言之，算术运算式可以嵌套，比如：1+（2*3）就是一个嵌套算术运算式，因为它的第二个运算量式（2*3）又是一个算术运算式，其运算量是 2 和 3，运算符号是*。两种运算量具有同样的接口，算术运算会一视同仁。

请用合成模式的语言解释这个例子。

（这个例子最早是由 Michael Duell 在文献[MDUELL97]中提出的，本书鼓励读者阅读

原文）

3. 请考察一个台式计算机的结构，并使用树结构的语言描述。

4. 请使用对象树的概念考察商业银行的 Liquidity 业务中的资金集中（Concentration）操作。

问答题答案

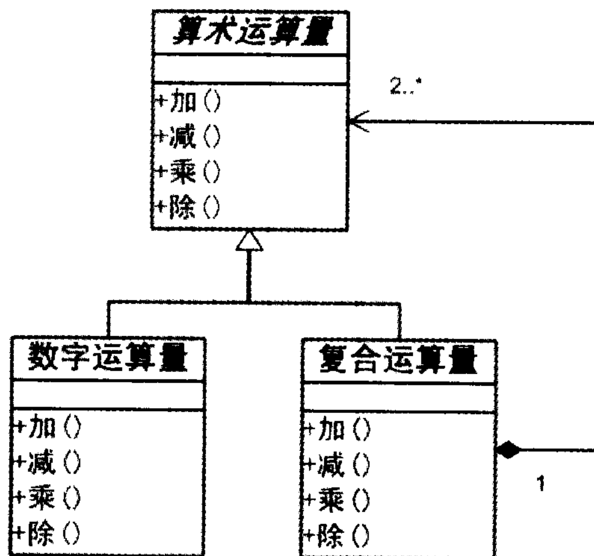
1. 这是一个很有意思的问题。很多初学者实际上把合成模式的定义类图错记成这个类图。

首先来看一看这个类图和合成模式的类图的区别在什么地方。在合成模式的类图里面，聚合的关系是从 Composite 到 Leaf 的，这意味着类型为 Composite 的对象可以含有类型为 Component 的对象。或者说，Composite 的对象不仅可以含有类型为 Leaf 的对象，而且可以含有类型为 Composite 的对象。

在本题给出的类图中，聚合的关系不是从 Composite 到 Component 的，而是从 Composite 到 Leaf 的。这就是说，类型为 Composite 的对象会含有 Leaf，这也就意味着它不会含有类型为 Composite 的对象。

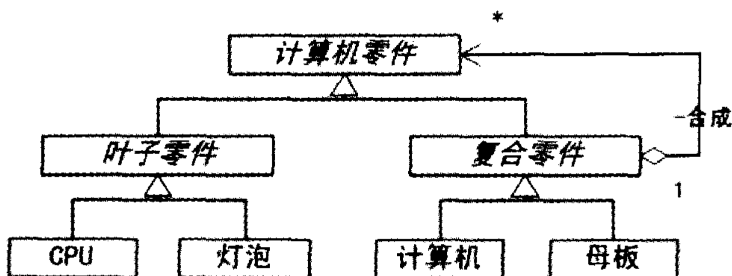
如果仍然以文件和目录的例子为例来解释的话，这时候的目录只有一层，也就是说目录只含有文件而不含有目录，目录的层次将只有一层。

2. 算术运算量所需的接口由一个抽象的“算术运算量”类给出。算术运算量有两种，一种是纯数字，相当于抽象的“算术运算量”类的树叶子类；另一种是合成算术量，本身由其他的运算量组成。算术运算的类图如下图所示。



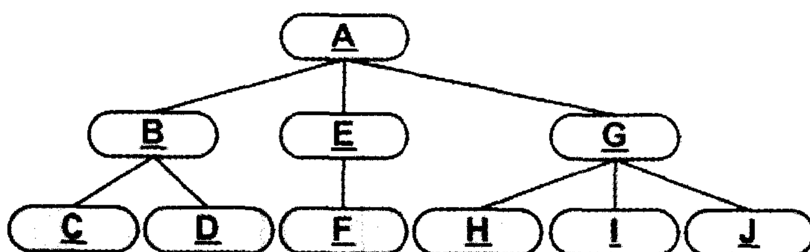
3. 一台计算机的内部结构可以用一个树结构描述。计算机的零件可以分为两种，即含有其他零件的树枝类零件，以及不含有其他零件的树叶类零件。

下图所示就是一个示意性的描述计算机零件的类图。



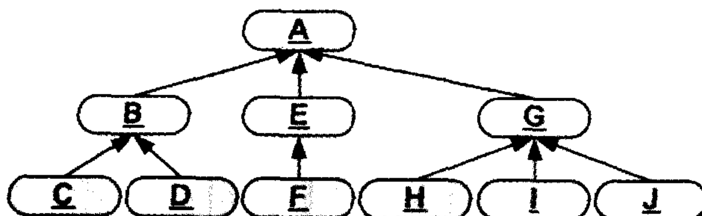
4. 商业银行的 Liquidity 业务是为了让客户可以最大限度地得到资金的使用率和回报率。

资金的集中要求将账号划分为几个层次，下图所示就是有两个层次的账号树结构。



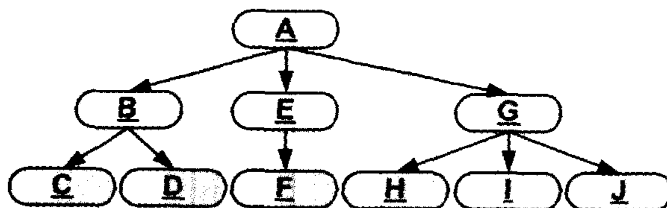
在一个工作日结束时，银行的系统会执行一个批处理，将账号 C 和 D 中的资金移动到账号 B 中，与 B 中原来就有的资金合在一起；账号 F 中的资金移动到账号 E，账号 H、I、J 中的资金移动到账号 G，与账号 E 和 G 中原有的资金合在一起；然后账号 B、E、G 中的资金继续向上移动到账号 A 中，与账号 A 中原来就有的资金合在一起。

下图所示的资金由下向上的有向树图就表示了每个工作日结束时的操作。



这样一来，客户的这些账号中所有的资金就全部集中到顶级账号 A 中，并通过适当管道将资金投入投资管道增值。增值部分以利息方式进入到账号 A 中。

在下一个工作日开始时，原来被移动到上面层次的账号中的资金会沿着一个相反的过程移动，从账号 A 移动到 B、E、G 中，在继续移动到账号 C、D、E、F、H、I、J 中。资金向下移动时的有向结构图如下图所示。



因此，资金集中的过程涉及到账号由上向下的树图结构和从下向上的树图。

参考文献

[MDUELL97] Michael Duell . Catalog of Non-Software Examples of Design Patterns. Object Magazine，July 1997

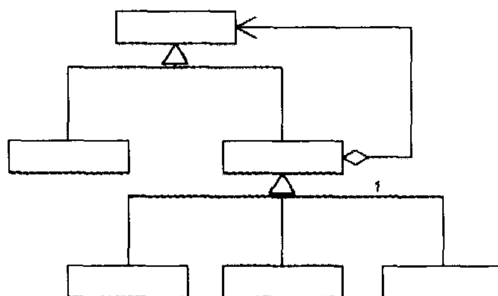
第 26 章　装饰（Decorator）模式

装饰（Decorator）模式又名包装（Wrapper）模式[GOF95]。装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。

26.1　引　　言

孙悟空有七十二般变化，他的每一种变化都给他带来一种附加的本领。他变成鱼儿时，就可以到水里游泳；他变成雀儿时，就可以在天上飞行。而不管悟空怎么变化，在二郎神眼里，他永远是那只猢狲。

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任。换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不使用创造更多子类的情况下，将对象的功能加以扩展。装饰模式的简略类图如下图所示。



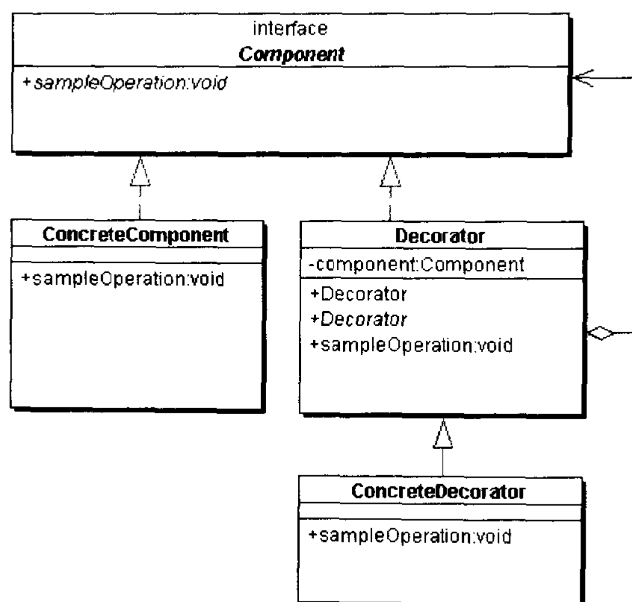
26.2　装饰模式的结构

装饰模式使用原来被装饰的类的一个子类的实例，把客户端的调用委派到被装饰类。装饰模式的关键在于这种扩展是完全透明的。

在孙猴子的例子里，老孙变成的鱼儿相当于老孙的子类，这条鱼儿与外界的互动要通过“委派”，交给老孙的本尊，由老孙本尊采取行动。尽管老孙把自己“装饰”成了鱼儿，在二郎神的眼里，他仍然是那只猢狲。

模式的类图

装饰模式的类图如下图所示。



在装饰模式中的各个角色有：

- 抽象构件（Component）角色：给出一个抽象接口，以规范准备接收附加责任的对象。
- 具体构件（Concrete Component）角色：定义一个将要接收附加责任的类。
- 装饰（Decorator）角色：持有一个构件（Component）对象的实例，并定义一个与抽象构件接口一致的接口。
- 具体装饰（Concrete Decorator）角色：负责给构件对象"贴上"附加的责任。

源代码

下面就给出装饰模式的示意性源代码。首先是抽象构件角色的源代码，如代码清单 1 所示。

代码清单 1：定义中的构件类的示意性源代码

```
package com.javapatterns.decorator;
public interface Component
{
    /**
     *  商业方法
     */
    void sampleOperation();
}
```

下面是装饰角色的源代码，如代码清单 2 所示。

代码清单 2：定义中的装饰类的示意性源代码

```
package com.javapatterns.decorator;
public class Decorator
    implements Component
```

```
{
    private Component component;
   /**
    *  构造子
    */
    public Decorator(Component component)
    {
        this.component = component;
    }
   /**
    *  构造子
    */
    public Decorator()
    {
        //write your code here
    }
   /**
    *  商业方法，委派给构件
    */
    public void sampleOperation()
    {
        component.sampleOperation();
    }
}
```

应当指出的有以下几点：

- 在上面的装饰类里，有一个私有的属性 component，其数据类型是构件（Component）。
- 此装饰类实现了构件（Component）接口。
- 接口的实现方法也值得注意，每一个实现的方法都委派给父类，但并不单纯地委派，而是有功能的增强。

虽然 Decorator 类不是一个抽象类，在实际应用中也不一定是抽象类，但是由于它的功能是一个抽象角色，因此也常常称它为抽象装饰。

下面就是具体装饰角色的源代码，如代码清单 3 所示。

代码清单 3：定义中的具体构件类的示意性源代码

```
package com.javapatterns.decorator;
public class ConcreteComponent
    implements Component
{
   /**
    *  构造子
    */
    public ConcreteComponent()
    {
```

```
            // Write your code here
        }
        /**
         *  商业方法
         */
        public void sampleOperation()
        {
            // Write your code here
        }
    }
```

具体装饰类实现了抽象装饰类所声明的 sampleOperation()方法，如代码清单 4 所示。

代码清单 4：定义中的具体装饰类的示意性源代码

```
package com.javapatterns.decorator;
public class ConcreteDecorator
    extends Decorator
{
    /**
     *  商业方法
     */
    public void sampleOperation()
    {
        super.sampleOperation();
    }
}
```

对象图

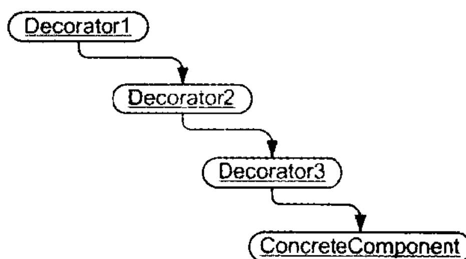
装饰模式的对象图呈链状结构，其共有三个具体装饰类，分别称为 Decorator1、Decorator2 和 Decorator3，具体构架类是 ConcreteComponent。一个典型的创建过程如代码清单 5 所示。

代码清单 5：一个典型的创建过程的源代码

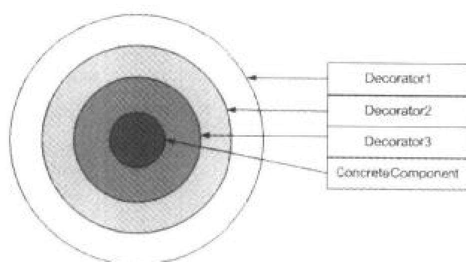
```
new Decorator1(
    new Decorator2(
        new Decorator3(
            new ConcreteComponent()
            )
        )
    );
```

这就意味着 Decorator1 的对象持有一个对 Decorator2 对象的引用，后者则持有一个对 Decorator3 对象的引用，再后者持有一个对具体构件 ConcreteComponent 对象的引用，这种链式的引用关系使装饰模式看上去像是一个 LinkedList，如下图所示。



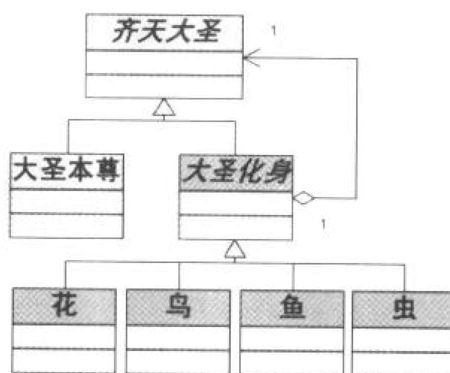


装饰模式常常被称做包裹模式，就是因为每一个具体装饰类都将下一个具体装饰类或者具体构件类包裹起来。仍然以上面的情况为例，Decorator1 对象包裹了 Decorator2 对象，后者包裹了 Decorator3 对象，再后者又包裹了 ConcreteComponent 对象。每一层包裹都提供了一些新的功能，如下图所示。



齐天大圣的例子

回到齐天大圣的例子，Component 的角色便由大名鼎鼎的齐天大圣孙悟空扮演；ConcreteComponent 的角色属于大圣的本尊，就是猢狲本人；大圣的七十二变扮演的便是 Decorator 角色；而 ConcreteDecorator 的角色便是花、鸟、鱼、虫等七十二般变化。孙大圣的装饰模式解释如下图所示。



其 Java 语言的源代码如下所示。首先是抽象构件角色“齐天大圣”接口的源代码，如代码清单 6 所示。

代码清单 6：抽象构件角色的源代码

```
// 大圣的尊号（大圣接口）
public interface 齐天大圣
```

```
{
    public 齐天大圣();
    public void move();
}
```

可以看出，“齐天大圣”定义出了一个 move()方法，这是所有的具体构件类和装饰类必须实现的。

下面是抽象装饰角色“七十二般变化”的源代码，如代码清单 7 所示。可以看出，它实现了抽象构件角色所要求的方法。

代码清单 7：抽象装饰角色的源代码

```
// 大圣的七十二般变化，抽象装饰
public class 七十二般变化
{
    private 齐天大圣 c;
    public 七十二般变化（齐天大圣 c）
    {
        super();
        this.c = c;
    }

    public void move()
    {
        c.move();
    }
}
```

下面是具体构件角色“大圣本尊”的源代码，如代码清单 8 所示。

代码清单 8：具体构件角色的源代码

```
// 大圣本尊类（实大圣类）
public class 大圣本尊 extends 齐天大圣
{
    public 大圣本尊()
    {
        // 代码
    }
    public void move()
    {
        // 代码
    }
}
```

下面是具体装饰角色“鱼”的源代码，如代码清单 9 所示，它实现了抽象装饰角色的接口。



代码清单 9：具体装饰模式角色的源代码

```
// 大圣的变化之一：实变化鱼类
public class 鱼 extends 七十二般变化
{
    public 鱼()
    {
        // 代码
    }
    public 鱼(齐天大圣 c)
    {
        // 代码
    }
    public void move()
    {
        super.move();
    }
}
```

具体装饰角色“花”、“鸟”、“虫”等的源代码与“鱼”的源代码大同小异，因此省略。

26.3　装饰模式应当在什么情况下使用

在以下情况下应当使用装饰模式：

（1）需要扩展一个类的功能，或给一个类增加附加责任。

（2）需要动态地给一个对象增加功能，这些功能可以再动态地撤销。

（3）需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变得不现实。

26.4　孙大圣的故事

用孙大圣的例子来说，首先你有一个大圣本尊的实例，即：

```
齐天大圣 c = new 大圣本尊();
```

然后，你把大圣“装饰”成一只雀儿：

```
齐天大圣 bird = new 雀儿( c );
```

由于“大圣本尊”是 ConcreteComponent 类，因此可以调用默认构造子创建实例，而“雀儿”是装饰类。要装饰的是“大圣本尊”，也即“猢狲”实例。换言之，上面的 Java 语句把“猢狲”变成了“雀儿”（把“雀儿”的功能加到了“猢狲”身上）。

由于多态性原则，“雀儿”构造子可以接受任何“齐天大圣”的子类的实例。这也就

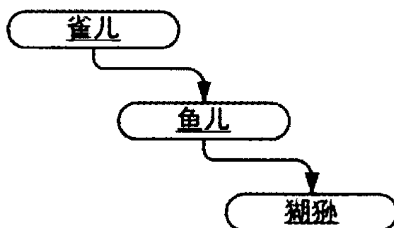
意味着可以把任何大圣的化身装饰成“雀儿”（把“雀儿”的功能加到大圣的任何化身之上）。比如：

```
齐天大圣  c = new 大圣本尊();
齐天大圣  fish = new 鱼儿( c );
齐天大圣 bird = new 雀儿( c );
```

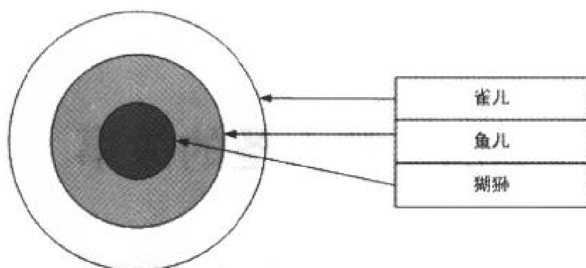
从而系统把大圣从一只猢狲装饰成了一条鱼儿（把鱼儿的功能加到了猢狲身上，得到了猢狲+鱼儿），然后又把猢狲装饰成了一只雀儿（把雀儿的功能加到了猢狲身上，得到了猢狲+雀儿），如下所示。

```
齐天大圣  c = new 大圣本尊();
齐天大圣  fish = new 鱼儿( c );
齐天大圣 bird = new 雀儿( bird );
```

从而系统把大圣从一只猢狲装饰成了一条鱼儿（把鱼儿的功能加到了猢狲身上），然后又把鱼儿装饰成了一只雀儿。（把雀儿的功能加到了猢狲+鱼儿身上，得到了猢狲+鱼儿+雀儿），如下图所示。



大圣的变化首先将鱼儿的功能附加到了猢狲身上，然后又将雀儿的功能附加到猢狲+鱼儿身上，如下图所示。



26.5 使用装饰模式的优点和缺点

使用装饰模式主要有以下的优点：

（1）装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。

装饰模式允许系统动态地决定“贴上”一个需要的“装饰”，或者除掉一个不需要的

“装饰”。继承关系则不同，继承关系是静态的，它在系统运行前就决定了。

这就如同孙大圣变成的鱼儿和鱼儿爸爸、鱼儿妈妈生出的鱼儿宝宝的不同。用二郎神的话说，“打花的鱼儿，似鲤鱼，尾巴不红；似鳜鱼，花鳞不见；似黑鱼，头上无星；似鲂鱼，腮上无针……必然是那猴变的。”老孙可以变成鱼儿，可以变成一只雀儿，也可以在瞬间变回猢狲去。而继承而来的鱼儿和雀儿都有爸爸和妈妈，有出生的历史，没有这等的变化本事。

（2）通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

这一点似乎和孙大圣的本领相仿。孙大圣的变化就可以排列组合，花样翻新。而继承关系则没有这个优势，每一种不同的排列组合均需要事先通过子类的继承方式设计好。

（3）这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错。

例如，不同的装饰类在排列组合时会产生不合理的或荒谬的组合，或是制造出循环引用的错误来。比如，孙大圣在和二郎神斗法时，变成的土地庙后面的旗杆便不太地道，暴露了狐狸（实际是猢狲）的尾巴，惹二郎神笑道：“是这猢狲了！他今又在那里哄我。我也曾见庙宇，更不曾见一个旗杆竖在后面的。”

使用装饰模式主要有以下的缺点：

由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

当然，如果程序员都如二郎神一般神勇，可以小圣降服大圣，查错料也无妨。

26.6　模式实现的讨论

模式的简化

大多数情况下，装饰模式的实现都比本节的定义中给出的示意性实现要简单。对模式进行简化时需要注意以下的情况：

（1）一个装饰类的接口必须与被装饰类的接口相容。

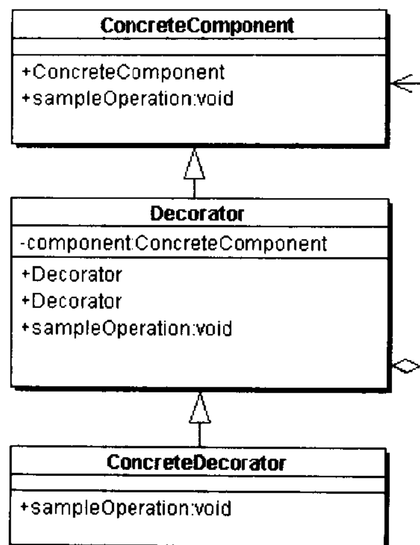
ConcreteDecorator 类必须继承自一个共同的父类 Component。这一点在本节早些的定义里是很清楚的。但是在实际使用时，如果在模式的实现上有所简化，就必须特别注意这一点。

（2）尽量保持 Component 作为一个“轻”类。

这个类的责任是为各个 ConcreteDecorator 类提供共同的接口，因此它应当着重在提供接口而不是存储数据。在本章早些时候给出的示意性图中，Component 干脆是一个 Java 接口；而在实际工作中，它可以是一个抽象类或是一个具体类。此时，就应当注意不要把太多的逻辑和状态放在 Component 类里。

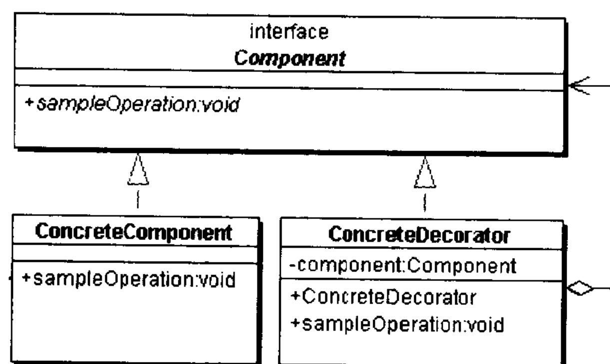
（3）如果只有一个 ConcreteComponent 类而没有抽象的 Component 类（接口），那么

Decorator 类经常可以是 ConcreteComponent 的一个子类，如下图所示。



由上图可知没有抽象的接口 Component 也是可以的，但 ConcreteComponent 就要扮演双重的角色。

（4）如果只有一个 ConcreteDecorator 类，那么就没有必要建立一个单独的 Decorator 类，而可以把 Decorator 和 ConcreteDecorator 的责任合并成一个类。甚至在只有两个 ConcreteDecorator 类的情况下，都可以这样做。但是如果 ConcreteDecorator 类的数目大于三的话，使用一个单独的 Decorator 类来区分抽象和具体的责任就是必要的了，如下图所示。



由上图可知没有抽象的 Decorator 也是可以的，只是 ConcreteDecorator 需要扮演双重的角色。

透明性的要求

最后，本书指出一个重要的实现问题，通常叫做针对抽象编程。装饰模式对客户端的透明性要求程序不要声明一个 ConcreteDecorator 类型的变量，而应当声明一个 Component 类型的变量。换言之，下面的做法是对的：



```
Component c = new ConcreteComponent();
Component c1 = new ConcreteDecorator1(c);
Component c2 = new ConcreteDecorator( c1 );
```

而下面的做法是不对的：

```
ConcreteComponent c = new ConcreteDecorator();
```

这就是前面所说的，装饰模式对客户端是完全透明的含义。

用孙悟空的例子来说，必须永远把孙悟空的所有变化都当成孙悟空来对待，而如果把老孙变成的雀儿当成雀儿，而不是老孙，那就被老孙骗了，而这是不应当发生的。

换言之，下面的做法是对的：

```
齐天大圣 c = new 大圣本尊();
```

或者：

```
齐天大圣 bird = new 雀儿( c );
```

而下面的做法是不对的：

```
大圣本尊 c = new 大圣本尊();
雀儿 bird = new 雀儿( c );
```

这就意味着在 ConcreteDecorator 里面不可以有 Component 里面所没有的方法。为什么呢？如果在 ConcreteDecorator 里面有一个新的方法，比如叫做 newMethod()，那么，客户端怎么调用这个 newMethod()呢？记住客户端只有 Component 接口，而这个接口里面并没有 newMethod()方法，因此像下面的语句根本过不了编译器这一关：

```
Component c = new ConcreteComponent();
Component c1 = new ConcreteDecorator(c);
c1.newMethod();
```

半透明的装饰模式

然而，纯粹的装饰模式很难找到。装饰模式的用意是在不改变接口的前提下，增强所考虑的类的性能。在增强性能的时候，往往需要建立新的公开的方法。即便是在孙大圣的系统里，也需要新的方法。比如齐天大圣类并没有飞行的能力，而雀儿有。这就意味着雀儿应当有一个新的 fly()方法。再比如，齐天大圣类并没有游泳的能力，而鱼儿有，这就意味着在鱼儿类里应当有一个新的 swim()方法。

这就导致了大多数的装饰模式的实现都是“半透明”（semi-transparent）的，而不是完全“透明”的。换言之，允许装饰模式改变接口，增加新的方法。这意味着客户端可以声明 ConcreteDecorator 类型的变量，从而可以调用 ConcreteDecorator 类中才有的方法：

```
齐天大圣 c = new 大圣本尊();
雀儿 bird = new 雀儿( c );
bird.fly();
```

齐天大圣接口根本没有 fly()这个方法，而雀儿接口里有这个方法。

但是，只要客户端不需要调用这些属于装饰的方法，而只调用属于 Component 的方法，那么装饰模式就仍然等同于透明的。

半透明的装饰模式是介于装饰模式和适配器模式之间的。适配器模式的用意是改变所考虑的类的接口，也可以通过改写一个或几个方法，或增加新的方法来增强或改变所考虑的类的功能。大多数的装饰模式实际上是半透明的装饰模式，这样的装饰模式也称做半装饰、半适配器模式。

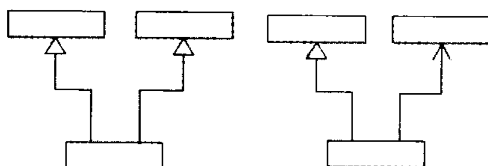
在模式研究圈子里，所谓半透明的装饰模式又叫做退化的（degenerate）装饰模式。

26.7 装饰模式与其他模式的关系

装饰模式与适配器模式、合成模式、策略模式等有相似之处和联系。

装饰模式与适配器模式的区别和联系

读者可能已经注意了，装饰模式和适配器模式都有一个别名，即包装（Wrapper）模式，但是这两个模式是很不一样的。适配器模式的用意是要改变所考虑的对象的接口而不一定改变对象的性能，而装饰模式的用意是要保持接口，从而增强所考虑对象的性能。然而，两者的关系并不止于此，如下图所示。

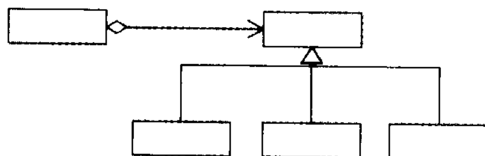


上图中左边是类的适配器模式，右边是对象的适配器模式。

装饰模式与策略模式的关系

装饰模式将一个东西的表皮换掉，而保持它的内心。策略模式恰好相反，它在保持接口不变的情况下，使具体算法可以互换。

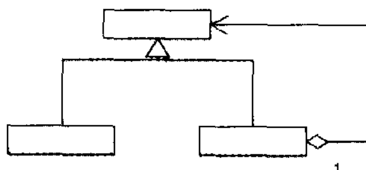
装饰模式的实现要求 Component 类尽量地“轻”，而策略模式要求抽象策略类尽量的“重”，策略模式的简略类图如下图所示。





装饰模式与合成模式的关系

装饰模式常常用在合成模式的行为扩展上。使用继承关系扩展合成模式的行为很困难。如果仅仅对抽象构件（Component）类还是合成类（Composite）类或者树叶（Leaf）类使用继承办法，都会导致多态性被破坏。惟一能够保持多态性的办法，便是对所有的三种角色都使用继承关系，而这显然不是很好的办法。合成模式的简略类图如下图所示。



装饰模式是继承关系的替代方案。装饰模式可以动态地为合成模式添加新的行为。关于合成模式，请参见“合成（Composite）模式”一章。

26.8　实例：GrepReader

这里给出一个实用的例子，以说明半透明的装饰模式的应用。

什么是 Grep

Grep 是 UNIX 操作系统中的命令，它的功能强大，可以用来处理对流的搜索。由于 Grep 给出的结果仍然是流，因此可以作为过滤器（filter）在任何一条管道线（pipeline）中使用。比如下面的命令在文件 file1 中搜索所有含有 BMW 字样的行，并显示出来：

```
Grep BMW file1
```

一个用 Java 语言实现的 Grep 系统

下面就用 Java 语言的 I/O 包来实现一个简单的搜索器，叫做 Grep，进行与上面的搜索相似的工作。为简单起见，这个小系统不处理正则表达式（Regular Expression）。

MVC 模式

首先必须把程序的功能做一个合理的划分，也即宏观的设计。这里把所有关于流的操作都放到一个叫 GrepReader 的类中，而显示输出数据的功能都放到一个叫 GrepView 的类中。这样就把这个系统按照 MVC（Model-View-Controller）的模式划分成模型（model）、视图（view）和控制器（control）三个层次：

- 模型（Model）：即 GrepReader 类，是一个对输入的文件流进行处理的构件。真正的搜索就发生在这里，顾名思义，这是一个处理 char 流的构件。

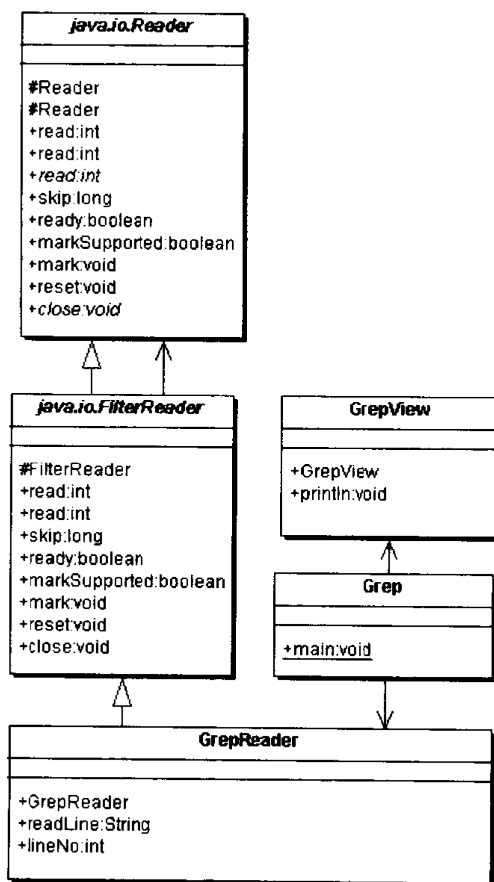
- 视图（View）：即 GrepView 类，是一个简单的数据输出工具。到目前为止，它只是把数据打印到屏幕上。
- 控制器（Control）：即 Grep 类，是系统的控制中心。

MVC 模式可以应用到系统的宏观以及微观尺度的设计上。在这个小系统里，MVC 模式被应用到了系统的总体功能的划分上，这属于宏观功能的划分。装饰模式与其他的各种设计模式一样，是为了系统的微观设计而准备的。关于 MVC 设计模式请见“专题：MVC 模式与用户输入数据检查”一章。

微观设计

Grep 系统的模型构件，即 GrepReader 有进行微观设计的必要。这里决定要 GrepReader 处理 char 流，这表明 GrepReader 属于 Reader 等级结构，应当继承自 Reader 抽象类或其子类。此外还要求 GrepReader 可以处理来自文件的流，而且输出必须也是一个流。这就是说，GrepReader 是一个过滤类，应当继承自 FilterReader 抽象类。

为了提供 Grep 系统所需要的功能，GrepReader 类必须增加一个 readLine()方法，一个 lineNo 性质，而且必须有一个构造子以接收一个 FileReader 类型的实例和一个要搜索的字符串，下图所示就是 Grep 系统的 UML 结构图。



这样就完成了系统的设计，而具体写代码的工作便可以交给初级程序员去完成了。实际上，这个工作已经完成了，如代码清单 10 所示。



代码清单 10：GrepReader 类的源代码

```
package com.javapatterns.decorator.Grepr;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.FilterReader;
public class GrepReader
    extends FilterReader
{
    // substring to be matched
    protected String substring;
    // InputReader
    protected BufferedReader in;
    private int lineNumber;

   /**
    *  构造子
    */
    public GrepReader( FileReader in,
        String substring )
    {
        super( in );
        this.in = new BufferedReader( in );
        this.substring = substring;
        lineNumber = 0;
    }
    public final String readLine( )
        throws IOException
    {
        String line;
        // step forward until line
        // matching substring
        do
        {
            line = in.readLine( );
            lineNumber++;
        } while ( (line != null) &&
            line.indexOf( substring ) == -1 );
        return line;
    }
    public int getLineNo( )
    {
        return lineNumber;
    }
}
```

在 GrepReader 类的源代码中可以看到，一个 FileReader 的实例被传进 GrepReader 的构造子里面。这个 FileReader 的实例被装饰成 BufferedReader 类（BufferedReader 是装饰类），并被传给父类的保护属性 in，注意 in 的数据类型是 Reader，如代码清单 11 所示。

代码清单 11：Grep 类的源代码

```
package com.javapatterns.decorator.Grepr;
import java.io.*;
public class Grep
{
    static GrepReader g;
    private static GrepView gv =
        new GrepView();
    public static void main( String[] args  )
    {
        String line;
        if ( args.length <= 1 )
        {
            gv.println("Usage: java Grep   ");
            gv.println("          no regexp");
            gv.println("          files to be searched in");
            System.exit( 1 );
        }
        try
        {
            gv.println( "\nGrep: 搜索 " + args[0] + " 文件 " + args[1] );
            gv.println( "文件和行号\t\t 下面的行里含有所搜索的字符串\n" );
            g = new
                new FileReader( args[1] ), args[0] );
            for( ; ; )
            {
                line = g.readLine( );
                if (line == null) break;
                gv.println( args[1] + ":"
                    + g.lineNo( ) + ":\t" + line );
            }
            g.close( );
        }
        catch ( IOException e )
        {
            gv.println( e.getMessage() );
        }
    }
}
```

GrepView 类的源代码如代码清单 12 所示。



代码清单 12：GrepView 的源代码

```
package com.javapatterns.decorator.Grepr;
import java.io.PrintStream;
public class GrepView
{
    PrintStream out ;
   /**
    *  构造子
    */
    public GrepView()
    {
        out = System.out;
    }
    public void println(String line)
    {
        out.println(line);
    }
}
```

在使用的时候，调用 Grep 类并传入要搜索的字符以及文件名。下图所示是系统运行时的情况。

```
C:\WINNT\System32\cmd.exe

C:\JavaPatterns\src>java com/javapatterns/decorator/grepr/Grep 猴 c:\monkeyking001.txt

Grep: 搜索 猴 文件 c:\monkeyking001.txt
文件行号                    下面的行里含有所搜索的字符串

c:\monkeyking001.txt:29:          似圆球样大。因见风，化作一个石猴，五官俱备，四肢皆
c:\monkeyking001.txt:32:          有一座花果山，山上有一仙石，石产一卵，见风化一石猴，
将潜息矣。”
c:\monkeyking001.txt:35:          那猴在山中，却会行走跳跃，食草木，饮涧泉，采山花，
c:\monkeyking001.txt:36:          夜宿石崖之下，朝游峰洞之中。真是“山中无甲子，寒尽

c:\monkeyking001.txt:40:          一群猴子耍了一会，却去那山涧中洗澡。见那股涧水奔流，
道：
c:\monkeyking001.txt:47:          众猴拍手称扬道：“好水！好水！原来此处远通山脚之下，
头出来.
```

程序运行结果给出在《西游记》的第一章里所有出现“猴”字的行。

26.9　一个例子：发票系统

要求

有一个电子销售系统需要打印出顾客所购买的商品的发票。一张发票可以分为三个部分：

- 发票头部（Header）：上面有顾客的名字，销售的日期。
- 发票主部：销售的货物清单，包括商品名字、购买的数量、单价、小计。

- 发票的尾部（Footer）：商品总金额。

下面就是要求打印出的发票的大致的样子。

```
    ***    INVOICE    ***
XYZ Incorporated
Date of Sale: Sun May 26 21:42:36 EDT 2002
==================================================
Item              Units    Unit Price    Subtotal
FireWheel Tire    4        ￥154.23      ￥616.92
Front Fender      1        ￥300.45      ￥300.45
==================================================
Total                                    ￥917.37
```

必须记住，发票的头部和尾部可以有很多种可能的格式，因此系统的设计必须给出足够的灵活性，使得一个新的头部和尾部格式能够较为容易地插入到系统中；同时本系统的客户端必须可以随意地选择某一个头部格式和某一个尾部格式的组合并与主部格式结合起来。

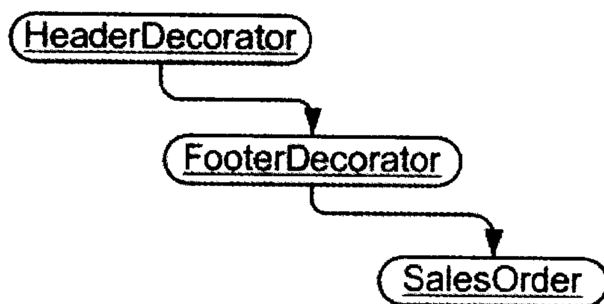
使用装饰模式

为了解决所面临的问题，必须选择一个合适的设计，使得系统可以处理很多的头部格式和尾部格式的组合，而这就暗示着设计师会考虑可以使用装饰模式。

在[GOF95]的书中是这样结实装饰模式的：（装饰模式）将更多的功能动态地附加到一个对象上。对功能扩展而言，装饰模式提供了一个灵活的、可以替代继承的选择。

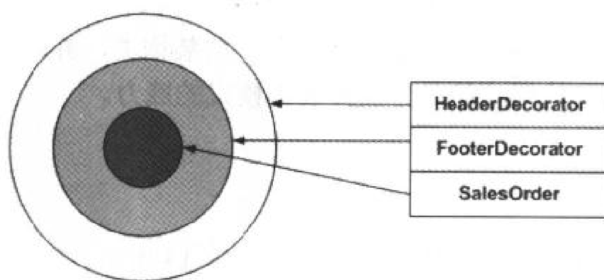
如果在这个系统里使用装饰模式，那么发票的头部和尾部都可以分别由具体装饰类 HeaderDecorator 和具体装饰类 FooterDecorator 来代表。有多少种头部和尾部，就可以有多少种相应的具体装饰类。这样一来，这些头部和尾部的组合就可以给出大量的选择。

正如本章的前面所述，装饰模式的对象图看上去很像是一个 LinkedList，每一个具体装饰对象都有一个引用指向下一个装饰对象或者是具体构件对象。在这里，一个 HeaderDecorator 对象指向一个 FooterDecorator 对象，后者指向 SalesOrder 对象，亦即具体构件角色。下图所示是发票系统的设计图。



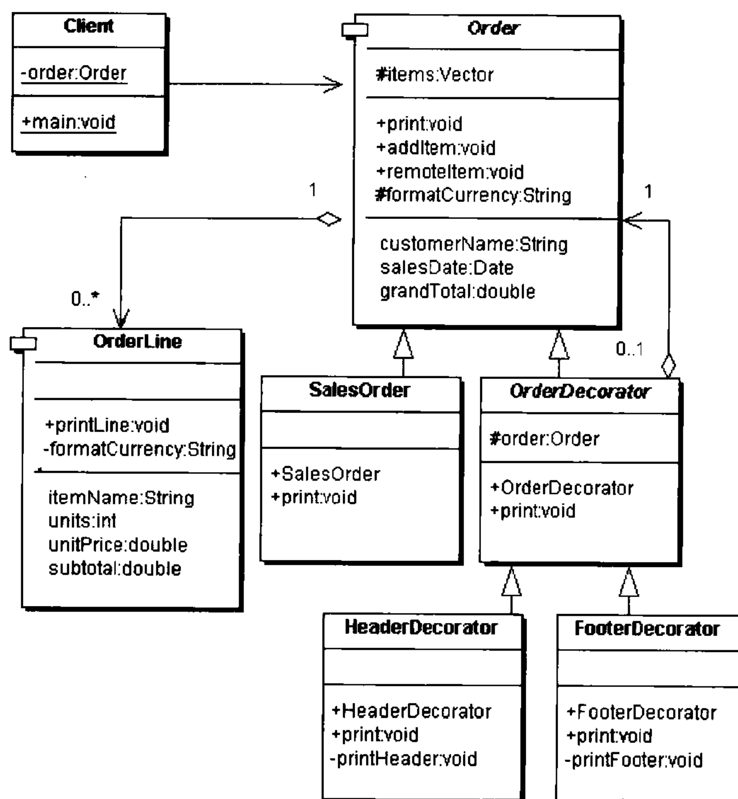
上面的对象图所描述的过程可以用包裹关系图清楚地描述，如下图所示。首先，HeaderDecorator 在包裹的最外层，然后是 FooterDecorator，最里面的是 SalesOrder。HeaderDecorator 层提供发票头部，而 FooterDecorator 提供了发票的尾部，最后 SalesOrder

则是发票的主部。



设计图

假设只有一个头部装饰类和一个尾部装饰类，那么发票系统的设计图如下图所示。



其中 SalesOrder 代表发票的主部，Order 是抽象构件角色，OrderDecorator 是抽象装饰角色。OrderLine 是一个发票的货品清单中的一行，给出产品名、产品单价、所购单位数、小计金额等。Order 类有一个 Vector 聚集，用以存储任意多个 OrderLine 对象。

HeaderDecorator 提供的打印功能比被装饰的对象的打印功能更强大，它除了调用被装饰的对象的 print()方法之外，还调用自己的私有方法 printHeader()方法，打印出发票的头部。

FooterDecorator 也是一样，除了调用被装饰的对象的 print()方法之外，还调用自己的

私有方法——printFooter()，打印出发票的尾部。

为了节省篇幅，在设计中仅给出了一个头部装饰类和一个尾部装饰类，但是读者可以根据自己系统的需要，加入更多的头部装饰类和尾部装饰类，并动态地选择头部装饰类和尾部装饰类的各种组合，这样才能充分发挥装饰模式的威力。

源代码

下面是抽象角色 Order 类的源代码，如代码清单 13 所示。

代码清单 13：抽象类 Order 的源代码

```
package com.javapatterns.decorator.printinvoice;
import java.util.Date;
import java.util.Vector;
import java.text.NumberFormat;
abstract public class Order
{
    private OrderLine lnkOrderLine;
    protected String customerName;
    protected Date salesDate;
    protected Vector items =
        new Vector(10);
    public void print()
    {
        for (int i = 0 ; i < items.size() ; i++)
        {
            OrderLine item =
                (OrderLine) items.get(i);
            item.printLine();
        }
    }
   /**
    *  客户名的取值方法
    */
    public String getCustomerName()
    {
        return customerName;
    }
   /**
    *  客户名的赋值方法
    */
    public void setCustomerName(
        String customerName)
    {
        this.customerName = customerName;
    }
```

```
/**
 *  销售日期的取值方法
 */
public Date getSalesDate()
{
    return salesDate;
}
/**
 *  销售日期的附值方法
 */
public void setSalesDate(Date salesDate)
{
    this.salesDate = salesDate;
}
/**
 *  增加一行销售产品
 */
public void addItem(OrderLine item)
{
    items.add(item);
}
/**
 *  删除一行销售产品
 */
public void remoteItem(OrderLine item)
{
    items.remove(item);
}
/**
 *  返还总额
 */
public double getGrandTotal()
{
    double amnt = 0.0D;
    for (int i = 0 ; i < items.size() ; i++)
    {
        OrderLine item = (OrderLine)
            items.get(i);
        amnt += item.getSubtotal();
    }
    return amnt;
}
/**
 *  工具方法，将金额格式化
 */
protected String formatCurrency(double amnt)
```

```
    {
        return NumberFormat.
            getCurrencyInstance().format(amnt);
    }
    }
```

具体构件类 SalesOrder 的源代码如代码清单 14 所示。

代码清单 14：具体类 SalesOrder 的源代码

```
package com.javapatterns.decorator.printinvoice;
public class SalesOrder extends Order
{
/**
 *  构造子
 */
public SalesOrder()
{
}
public void print()
{
    super.print();
}
}
```

抽象装饰角色 OrderDecorator 的源代码如代码清单 15 所示。

代码清单 15：抽象类 OrderDecorator 的源代码

```
package com.javapatterns.decorator.printinvoice;
abstract public class OrderDecorator
    extends Order
{
protected Order order;
/**
 *  构造子
 */
public OrderDecorator(Order order)
{
    this.order = order;
    this.setSalesDate( order.getSalesDate() );
    this.setCustomerName(
        order.getCustomerName() );
}
public void print()
{
    super.print();
}
}
```



可以看出，OrderDecorator 是依赖于被装饰对象的。它的构造子首先需要从被装饰对象中提取 salesOrder 和 customerName 等性质。一个更仔细的做法是对传入的参数加以检查，看是不是 null。作为一个示意性源代码，这里没有做这一检查。

下面是具体装饰类 HeaderDecorator 的源代码，如代码清单 16 所示。

代码清单 16：具体装饰类 HeaderDecorator 的源代码

```
package com.javapatterns.decorator.printinvoice;
public class HeaderDecorator
    extends OrderDecorator
{
  /**
   *  构造子
   */
   public HeaderDecorator(Order anOrder)
   {
       super(anOrder);
   }
   public void print()
   {
       this.printHeader();
       super.order.print();
   }
   private void printHeader()
   {
       System.out.println("\t***\tI N V O I C E\t***");
       System.out.println("XYZ Incorporated\nDate of Sale: ");
       System.out.println(order.getSalesDate());
       System.out.println("===========================");
       System.out.println("Item\t\tUnits\tUnit Price\tSubtotal");
   }
}
```

可以看出，HeaderDecorator 类提供了对打印功能的改进。除了调用被装饰对象的打印功能之外，还提供了私有的打印方法 printHeader()。

具体装饰类 FooterDecorator 的源代码如代码清单 17 所示。

代码清单 17：具体装饰类 FooterDecorator 的源代码

```
package com.javapatterns.decorator.printinvoice;
import java.text.NumberFormat;
public class FooterDecorator
    extends OrderDecorator
{
  /**
   *  构造子
   */
   public FooterDecorator(Order anOrder)
```

```
    {
        super(anOrder);
    }
    public void print()
    {
        super.order.print();
        printFooter();
    }
    private void printFooter()
    {
        System.out.println("========================");
        System.out.println("Total\t\t\t\t" +
            formatCurrency(super.order.getGrandTotal()));
    }
}
```

与 HeaderDecorator 一样，FooterDecorator 类提供了对打印功能的改进。除了调用被装饰对象的打印功能之外，还提供了私有的打印方法 printFooter()。

OrderLine 是一个销售单的货物清单中的一行，它包括下面的这些信息：产品名、数目、单价、小计金额，如代码清单 18 所示。

代码清单 18：OrderLine 类的源代码

```
package com.javapatterns.decorator.printinvoice;
import java.text.NumberFormat;
public class OrderLine
{
    private String itemName;
    private int units;
    private double unitPrice;
   /**
    *  产品名字的取值方法
    */
    public String getItemName()
    {
        return itemName;
    }
   /**
    *  产品名字的赋值方法
    */
    public void setItemName(String itemName)
    {
        this.itemName = itemName;
    }
   /**
    *  单位数量的取值方法
    */
```

```
    public int getUnits()
    {
        return units;
    }
   /**
    *  单位数量的附值方法
    */
    public void setUnits(int units)
    {
        this.units = units;
    }
   /**
    *  单价的取值方法
    */
    public double getUnitPrice()
    {
        return unitPrice;
    }
   /**
    *  单价的赋值方法
    */
    public void setUnitPrice(double unitPrice)
    {
        this.unitPrice = unitPrice;
    }
    public void printLine()
    {
        System.out.println(
            itemName + "\t" + units
            + "\t" + formatCurrency(unitPrice)
            + "\t" + formatCurrency(getSubtotal()));
    }
   /**
    *  小计金额的取值方法
    */
    public double getSubtotal()
    {
        return unitPrice * units;
    }
   /**
    *  工具方法，将金额格式化
    */
    private String formatCurrency(double amnt)
    {
        return NumberFormat.
            getCurrencyInstance().format(amnt);
```

```
    }
}
```

下面给出一个示意性的客户端的源代码，如代码清单 19 所示。

代码清单 19：Client 类的源代码

```
package com.javapatterns.decorator.printinvoice;
import java.util.Date;
public class Client
{
private static Order order;
public static void main(String[] args)
{
    order = new SalesOrder();
    order.setSalesDate(new Date());
    order.setCustomerName("XYZ Repair Shop");
    OrderLine line1 = new OrderLine();
    line1.setItemName("FireWheel Tire");
    line1.setUnitPrice(154.23);
    line1.setUnits(4);
    order.addItem(line1);
    OrderLine line2 = new OrderLine();
    line2.setItemName("Front Fender");
    line2.setUnitPrice(300.45);
    line2.setUnits(1);
    order.addItem(line2);
    order = new HeaderDecorator(
        new FooterDecorator(order));
    order.print();
}
}
```

在上面给出的这个示意性的客户端里，顾客购买了四个汽车轮胎，一个汽车前侧挡板，然后由系统打印出发票。

从上面的客户端代码可以看出，客户端并不能区分被装饰过的对象和没有被装饰之前的具体构件对象，这实际上是装饰模式的用意之一。

本书给出的这个例子受到了文献[SHALL01]的一个类似的例子的启发，本书鼓励读者阅读[SHALL01]原书。

问答题

1. 请用 UML 状态图描述二郎神与孙大圣斗法中两者的变化。

2. 张明敏的《我的一颗中国心》歌曲里面唱："洋装虽然穿在身，我的心仍是一颗中国心……"。请解释这个歌词里蕴含的是装饰模式还是策略模式？

3. 根据前面的讨论，抽象的 Component 类可以省略，见 Component 类图。请给出类

图中对应的所有 Java 语言代码。

4. 根据前面的讨论，抽象的 Decorator 可以省略，见 Decorator 类图。请给出类图中对应的所有 Java 语言代码。

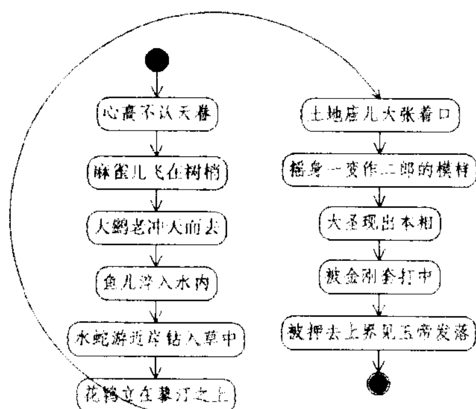
5. 根据前面的讨论，抽象的 Component 类可以省略。请按照这一方法，示范性地实现孙大圣的系统。

6. 根据前面的讨论，抽象的 Decorator 类可以省略。请问这是否适用于孙大圣的系统？

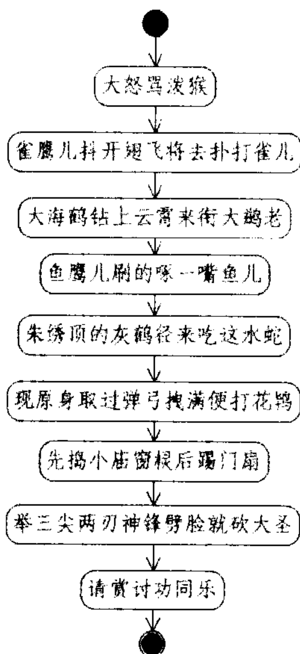
7. 如何在 Java 语言中动态地撤销装饰上去的功能？

问答题答案

1. 孙大圣的变化过程的 UML 状态图如下图所示。



二郎神的状态变化如下图所示。



2. 歌词里蕴含的是装饰模式。主人公的外表装饰改变了，而内心没有变化，这符合装饰模式的定义。而策略模式讲的是内心变化而外表没有变，更像是叛变了以后回来当特

务的，因此与歌词不符。

然而，在使用任何类比时一定要注意，没有一个类比是完全贴切的。

3. 省略了抽象构件角色之后，装饰模式的示意性实现的全部代码如下。首先是具体构件类，如代码清单 20 所示，这个类声明并自己实现了商业方法。

代码清单 20：具体构件类 Component 的源代码

```
public class ConcreteComponent
{
    public void sampleOperation()
    {
        // Write your code here
    }
}
```

抽象装饰角色继承自具体构件角色，因此它可以置换掉具体构件角色所声明的商业方法。事实上，如果一个装饰角色有实际意义的话，它就应当置换掉具体构件角色的商业方法，以便提供额外的功能，如代码清单 21 所示。

代码清单 21：装饰类 Decorator 的源代码

```
public class Decorator extends ConcreteComponent
{
    private ConcreteComponent component;
   /**
    *  构造子
    */
    public Decorator(ConcreteComponent
        component)
    {
        this.component = component;
    }
   /**
    *  构造子
    */
    public Decorator()
    {
        //write your code here
    }
    public void sampleOperation()
    {
        component.sampleOperation();
    }
}
```

具体装饰角色的源代码如代码清单 22 所示。可以看出，具体装饰角色置换掉了继承自具体构件角色的商业方法，从而达到增强具体构件角色的功能的目的。



代码清单 22：具体装饰类 ConcreteDecorator 的源代码

```
public class ConcreteDecorator
    extends Decorator
{
    public void sampleOperation()
    {
        super.sampleOperation();
    }
}
```

4. 在省略抽象装饰角色之后，系统的全部代码列出如下。

首先是抽象构件角色的源代码，如代码清单 23 所示。这个角色声明出所有的具体构件角色和装饰角色必须实现的接口，在这里以 sampleOperation()方法代表。

代码清单 23：抽象角色 Component 接口的源代码

```
public interface Component
{
    void sampleOperation();
}
```

由于抽象装饰角色被省略了，因此具体装饰角色自己要担负起抽象装饰角色和具体装饰角色的双重责任：一方面它要声明所有的接口，另一方面要实现这些接口。具体装饰角色的源代码如代码清单 24 所示。

代码清单 24：具体装饰角色的源代码

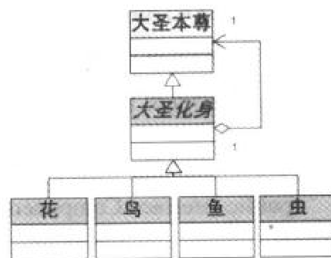
```
public class ConcreteDecorator
    implements Component
{
    private Component component;
   /**
    *  构造子
    */
    public ConcreteDecorator(
        Component component)
    {
        super();
        this.component = component;
    }
    public void sampleOperation()
    {
        component.sampleOperation();
    }
}
```

具体构件角色的源代码如代码清单 25 所示。这个角色需要实现抽象构件角色所声明的接口，在这里以 sampleOperation()方法代表。

代码清单 25：具体构件角色的源代码

```
public class ConcreteComponent
    implements Component
{
    public void sampleOperation()
    {
        // Write your code here
    }
}
```

5. 经过抽象的 Component 类后，大圣模拟系统的类图如右图所示。



可见，大圣的尊号没有了，因为抽象类“齐天大圣”类被省略了。

6. 由于孙大圣有七十二般变化，远多于一或二，因此省略抽象的 Decorator 的办法不适用于孙大圣系统。

7. 装饰类对象会自动地被 Java 垃圾收集器收集。

26.10 附录：关于适配器模式与装饰模式的对话

题材

这是一段关于桥梁模式、适配器模式和装饰模式的对话。对话中给出了学习模式的新手学生甲、学生乙向模式教师甲、教师乙请教 Java 语言的库 java.io 中所使用的模式。几个模式研究者和学习者在对话中互相补充，澄清了三个模式的类似之处和不同之处。

对话

学生甲：我有一个关于 java.io.InputStreamReader 类的问题。Java 语言文档说：

“InputStreamReader 是从 byte 流到字符（character）流的转换桥梁。它读入 byte 数据并根据所指定的字符流的编码规则翻译成字符数据……”

请问，这个 InputStreamReader 类是否是桥梁模式？

学生乙：桥梁模式的用意是把抽象化与实例化分开。我在这里看不到这样的桥梁。

教师甲：一个 InputStreamReader 类型的对象读入 byte 流，一个 Reader 类型的对象读入字符（character）流。而 InputStreamReader 是一种 Reader 类型，它把一个 InputStream 类型的对象包装起来，从而把 byte 流的 API 转换成字符流的 API。这是适配器模式的例子。InputStreamReader 把 InputStream 的 API 适配成 Reader 的 API。

因此，如果一个 InputStream 的子类适合所做的工作，但是用户端期待一个 Reader 的接口，那么就可以使用 InputStreamReader 把一个 InputStream 包装起来使用。



所以 Java 文档应该把它描写为适配器，而不是桥梁。

学生甲：现在我明白了。这个 InputStreamReader 类不是桥梁模式而是适配器模式。Java 语言文档并没有说这个类是一个桥梁模式，而只是说它是一个桥梁。然而，这样讲还是会有人认为它和桥梁模式有什么关系。

学生乙：我查了查 Java 文档，那上面写着这个类是 JDK 1.1 版本引进来的。那时候[GOF95]的著作刚刚发表，模式的名字还没有普遍推广。

Java 文档上的桥梁字眼与桥梁模式的名字没有关系，那里的桥梁仅仅指一种联系而已。

这里的 InputStreamReader 类应当是适配器类。不过，在很久之前我研究过 Java 的 IO 库，我记得那里所有的 Stream 类都是按照装饰模式类建造的。

教师甲：装饰模式和适配器模式都是包装（Wrapper）模式。它们之间的区别是，适配器模式把一个 API 转换成另一个 API，而装饰模式是保持被包装的对象的 API。用 Java 术语来讲，适配器和被适配的类实现的是不同的接口和抽象类，而装饰模式和被装饰的类实现的是不同的接口和抽象类。

InputStreamReader 是一个适配器，因为它把 InputStream 的 API 转换成 Reader 的 API。

在 java.io 库中，BufferedReader 是一个装饰类，因为它实现 Reader，并且包装（wrap）一个 Reader。类似地，BufferedInputStream、OutputStream、Writer 各自都是它们自己的装饰类。LineNumberReader、FilterReader 和 PushbackReader 均是 Reader 的装饰类，因为它们自己是 Reader 类，而且包装其他的 Reader 类。CharArrayReader、FileReader、PipedReader 和 StringReader 类不是装饰类，因为它们包装的是字符数值组、File、PipedWriter 和 String 类。它们应该被看做字符数值组、File、PipedWriter 和 String 类的适配器类。

这个例子显示 java.io 库结构包含有多个模式的应用。

教师乙：BufferedReader 是一个退化的装饰类，你也可以把它看成一个半装饰类、半适配器类，因为它实现了像 readLine()这样的不属于 Reader 接口的方法。

学生甲：为什么 BufferedReader 是一个退化的装饰类……？请再解释一下。

教师甲：教师乙提到了一个非常重要的观点。也就是说，很难找到一个纯而又纯的装饰类。以 BufferedReader 为例，客户端应当把它当做一个 Reader 类，而根本感觉不到 BufferedReader 的存在。用户端变量的数据类型仅仅应当是 Reader，从而客户端只能拿到 Reader 的方法。

正如教师乙所指出的，BufferedReader 还有一个 readLine()方法，而这个方法是 Reader 类的接口所没有的。如果客户端变量数据类型是 Reader，那么客户端怎么调用这个方法呢？反过来，如果客户端变量是 BufferedReader，那么客户端实际上知道它使用的是一个装饰，而这就破坏了装饰模式的原来的意图，装饰模式也就从装饰模式变成了适配器模式。在适配器模式里，适配器类的 API 和被适配的 API 会有很多重叠，但往往并不全同。

类似的例子在 VisualWork SmallTalk 的可视构件框架中也可以看到。那里大量的装饰类都有一个叫做 getSubject()的方法，以便客户端可以拿到被装饰的类，并调用装饰类所不接收的方法。客户端不应该知道装饰的存在，而调用 getSubject()方法的时候，客户端显然知道它调用的是一个装饰。

所以，BufferedReader 是一个装饰类，但它不是一个百分之百的装饰类，因为它提供

了一个 readLine()的新方法。一个装饰类实现的新方法越多，它离纯装饰类的距离也就越远。这也是模式理论遇到真实系统时会发生的事。

问答题

学生甲的关于 java.io.InputStreamReader 类是什么模式的问题，表明学生甲确实开始用模式的观点来看问题了。学生乙的问题确实是一个很好的问题，它还带出了 Java 语言的原始设计者在设计 Java 的 IO 库时都使用了什么设计模式的问题。

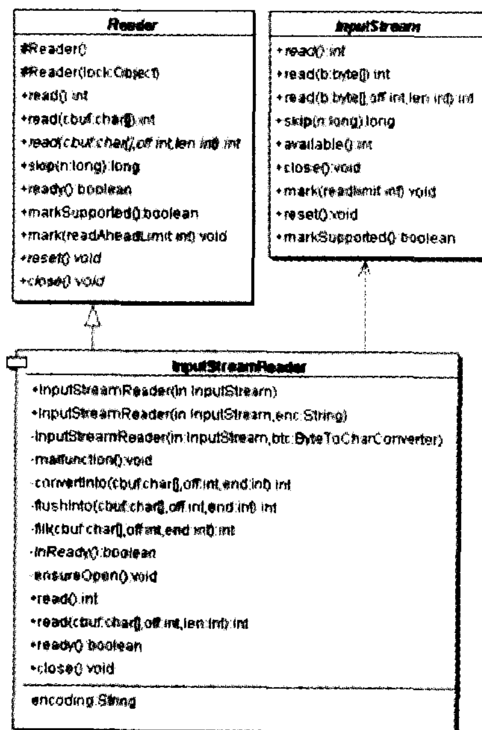
请读者试着回答下面的问题：

1. 请使用 UML 类图解释为什么 InputStreamReader 是适配器模式？

2. 请使用 UML 类图解释为什么 CharArrayReader 和 StringReader 是对象形式的适配器模式？

问答题答案

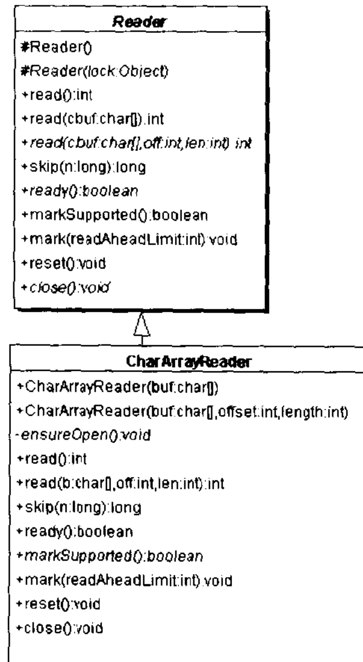
1. 如同在本书的“适配器（Adapter）模式”一章所讲的，与类形式的适配器模式一样，对象形式的适配器模式把被适配类的 API 转换成为目标类的 API。与类形式的适配器模式不同的是，对象形式的适配器模式不是使用继承关系连接到 Adaptee 类，而是使用委派关系连接到 Adaptee 类。InputStreamReader 恰好满足对象形式的适配器模式的定义，如下图所示。



从上图可以看出，InputStream 是被适配的类，而 Reader 是适配的目标类。InputStream Reader 做为适配器类把 InputStream 类的一个实例包装起来，从而能够把 InputStream 的 API

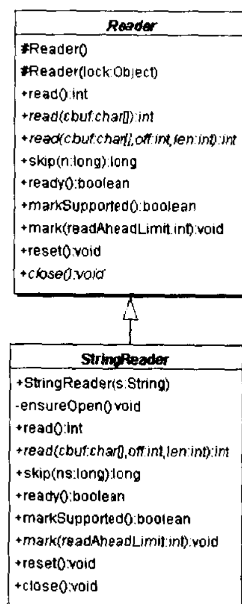
转换成为 Reader 的 API。

2. 正如教师甲所说的，CharArrayReader 实际上是一个对象形式的适配器模式。它把一个 Character 数组的实例包装起来，从而把 Character 数组的 API 与 Reader 类的 API 衔接起来。Character 数组就是 Adaptee，而 Reader 就是目标类，如下图所示。



CharArrayReader 是适配器类，它把 Character 数组的接口适配到 Reader 类的接口上。

StringReader 则是相似的，它把 String 对象的接口适配到 Reader 类的接口上，如下图所示。



从图中可以看到，与经典的对象形式的适配器模式不同，Adaptee 的实例是从构造子

处作为参量传进适配器类的，这并不改变这是一个适配器模式的事实。

参考文献

[SHALL01] Alan Shalloway. James R. Trott. Design Patterns Explained – A New Perspective on Object-Oriented Design. Addison-Wesley, 2001

第 27 章　专题：设计模式在 Java I/O 库中的应用

请读者在阅读了本书的“装饰（Decorator）模式”、“适配器（Adapter）模式”等章节后再阅读本章。本章的讨论是建立在 JDK 1.3 版的基础之上的。

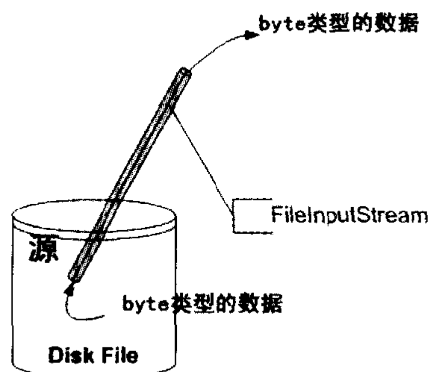
在 Java 语言 I/O 库的设计中，使用了两个结构模式，即装饰模式和适配器模式。本章就围绕这两个模式讨论 Java I/O 库的设计。

27.1 引　言

在任何一种计算机语言中，输入/输出都是一个很重要的部分。与一般的计算机语言相比，Java 将输入/输出的功能和使用范畴做了很大的扩充。因此，输入/输出在 Java 语言中占有极为重要的位置。

Java 语言采用流的机制来实现输入/输出。所谓流，就是数据的有序排列，而流可以是从某个源（称为流源，或者 Source of Stream）出来，到某个目的地（称为流汇，或者 Sink of Stream）去的。根据流的方向可以将流分成输出流和输入流，一个程序从输入流读取数据，而向输出流写出数据。

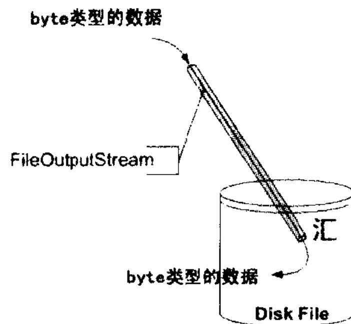
例如，一个 Java 程序可以使用 FileInputStream 类从一个磁盘文件读取数据，如下图所示。



像 FileInputStream 这样的处理器叫做流处理器。一个流处理器就像一个流的管道一样，从一个流源吸入某种类型的数据，并输出某种类型的数据。上面的这种示意图叫做流的管道图。

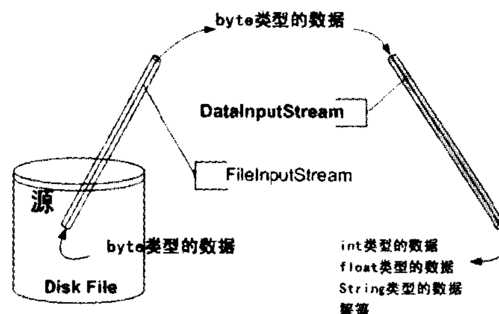
类似地，也可以使用 FileOutputStream 类向一个磁盘文件写出数据，如下图所示。





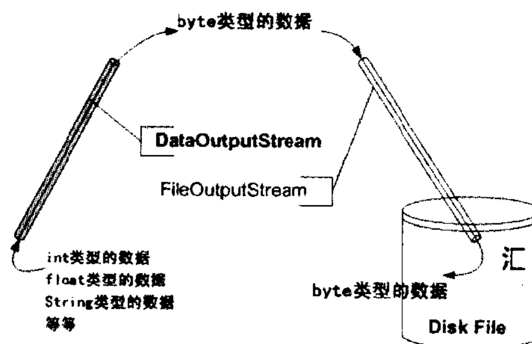
在实际应用当中，这样的简单机制并没有太大的用处。程序需要写出的往往是非常结构化的信息，因此这些 Byte 类型的数据实际上是一些数值、文字、源代码等。Java 的 I/O 库提供了一个称做链接（Chaining）的机制，可以将一个流处理器与另一个流处理器首尾相接，以其中之一的输出为输入，形成一个流管道的链接。

例如，DataInputStream 流处理器可以把 FileInputStream 流对象的输出当做输入，将 Byte 类型的数据转换成 Java 的原始类型和 String 类型的数据，如下图所示。



类似地，向一个文件写入 Byte 类型的数据不是一个简单的过程。一个程序需要向一个文件里面写入的数据往往都是结构化的，而 Byte 类型则是原始的类型。因此，在写的时候必须首先经过转换。DataOutputStream 流处理器提供了接收原始数据类型和 String 数据类型的方法，而这个流处理器的输出数据则是 Byte 类型。换言之，DataOutputStream 可以将源数据转换成 Byte 类型的数据，再输出出来。

这样一来，就可以将 DataOutputStream 与 FileOutputStream 链接起来。这样做的结果就是，程序可以将原始数据类型和 String 类型的源数据写到这个链接好的双重管道里面，达到将结构化数据写到磁盘文件里的目的，如下图所示。





当然，这又是链接的威力。

流处理器所处理的流必定都有流源，而如果将流类所处理的流源分类的话，那么基本可以分成两大类：

- 数组、String、File 等，这一种叫做原始流源。
- 同样类型的流用做链接流类的流源，就叫做链接流源。

关于这两种流源，下面还有更详细的介绍。

27.2　Java I/O 库的设计原则

Java 语言的 I/O 库是对各种常见的流源、流汇以及处理过程的抽象化。客户端的 Java 程序不必知道最终的流源、流汇是磁盘上的文件还是一个数组，或者是一个线程；也不必插手到诸如数据是否是经过缓冲的、可否按照行号读取等处理的细节中去。

第一次见到 Java I/O 库的人，无不因为这个库的庞杂而感到困惑；而熟悉这个库的人，又常常为这个库的设计是否得当而争论不休。要理解 Java I/O 这个庞大而复杂的库，关键是掌握两个对称性和两个设计模式。

Java I/O 库的两个对称性

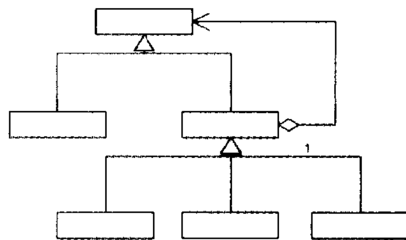
Java I/O 库具有两个对称性，它们分别是：

- 输出–输入对称：比如 InputStream 和 OutputStream 各自占据 Byte 流的输入与输出的两个平行的等级结构的根部；而 Reader 和 Writer 各自占据 Char 流的输入与输出的两个平行的等级结构的根部。
- byte–char 对称：InputStream 与 Reader 的子类分别负责 Byte 和 Char 流的输入；OutputStream 与 Writer 的子类分别负责 Byte 和 Char 流的输出，它们分别形成平行的等级结构。

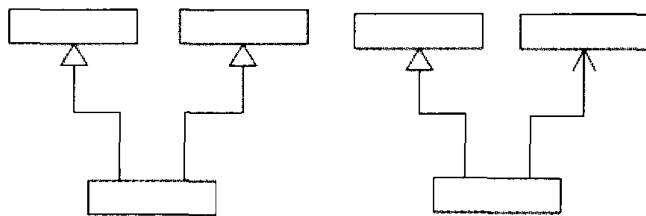
Java I/O 库的两个设计模式

Java I/O 库的总体设计是符合装饰模式和适配器模式的。如前所述，这个库中处理流的类叫做流类。文章开始的时候所谈到的 FileInputStream、FileOutputStream、DataInput Stream 以及 DataOutputStream 都是流处理器的例子。

- 装饰模式：在由 InputStream、OutputStream、Reader 和 Writer 代表的等级结构内部，有一些流处理器可以对另一些流处理器起到装饰作用，形成新的、具有改善了的功能的流处理器。装饰模式是 Java I/O 库的整体设计模式。这一个设计原则是符合装饰模式的，如右图所示。



- 适配器模式：在由 InputStream、OutputStream、Reader 和 Writer 代表的等级结构内部，有一些流处理器是对其他类型的流源的适配。这就是适配器模式的应用，如下图所示。



适配器模式应用到了原始流处理器的设计上面，构成了 I/O 库所有流处理器的起点。

27.3 装饰模式的应用

装饰模式在 Java 语言中的最著名的应用莫过于 Java I/O 标准库的设计了。本节就以 Byte 流的处理为例，讲解装饰模式是怎样应用到 Java I/O 库的设计中去的。

为什么要在 Java I/O 库中使用装饰模式

由于 Java I/O 库需要很多性能的各种组合，如果这些性能都是用继承的方法实现的，那么每一种组合都需要一个类，这样就会造成大量性能重复的类出现。而如果采用装饰模式，那么类的数目就会大大减少，性能的重复也可以减至最少。因此装饰模式是 Java I/O 库的基本模式。

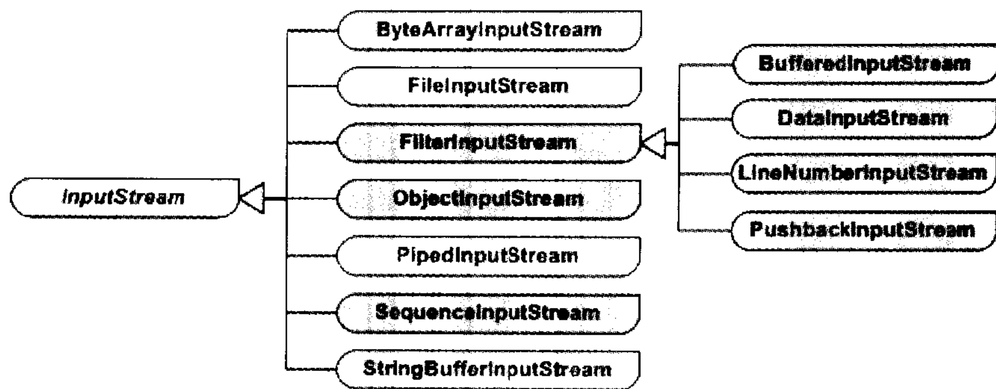
由于装饰模式的引进，造成灵活性和复杂性大大增加。在使用 Java I/O 库时，必须理解 Java I/O 库是由一些基本的原始流处理器和围绕它们的装饰流处理器所组成的，这样可以在学习和使用 Java I/O 库时做到事半功倍的效果。

InputStream 类型中的装饰模式

本节分析 InputStream 类型中的各个子类的功能和彼此之间的关系，以说明装饰模式是怎样应用到这个类型上去的。

InputStream 类型有哪些子类

InputStream 有七个直接的具体子类，有四个属于 FilterInputStream 的具体子类，如下图所示。



上图中所有的类都叫做流处理器，这个图就叫做（InputStream 类型的）流处理器图。根据输入流的源的类型，可以将这些流类分成两种，即原始流类（Original Stream）和链接流处理器（Wrapper Stream）。

原始流处理器

原始流处理器接收一个 Byte 数组对象、String 对象、FileDescriptor 对象或者不同类型的流源对象（也就是前面所说的原始流源），并生成一个 InputStream 类型的流对象。在 InputStream 类型的流处理器中，原始流处理器包括以下四种：

- ByteArrayInputSteam：为多线程的通信提供缓冲区操作功能，接收一个 Byte 数组作为流的源。
- FileInputStream：建立一个与文件有关的输入流。接收一个 File 对象作为流的源。
- PipedInputStream：可以与 PipedOutputStream 配合使用，用于读入一个数据管道的数据。接收一个 PipedOutputStream 作为源。
- StringBufferInputStream：将一个字符串缓冲区转换为一个输入流。接收一个 String 对象作为流的源。

与原始流处理器相对的是链接流处理器。

链接流处理器

所谓链接流处理器，就是可以接收另一个（同种类的）流对象（也就是前面所说的链接流源）作为流源，并对之进行功能扩展的类。InputStream 类型的链接流处理器包括以下几种，它们都接收另一个 InputStream 对象作为流源。

（1）FilterInputStream 称为过滤输入流，它将另一个输入流作为流源。这个类的子类包括以下几种：

- BufferInputStream：用来从硬盘将数据读入到一个内存缓冲区中，并从此缓冲区提供数据。
- DataInputStream：提供基于多字节的读取方法，可以读取原始数据类型的数据。
- LineNumberInputStream：提供带有行计数功能的过滤输入流。
- PushbackInputStream：提供特殊的功能，可以将已经读取的字节“推回”到输入流中。

（2）ObjectInputStream 可以将使用 ObjectOutputStream 串行化的原始数据类型和对象

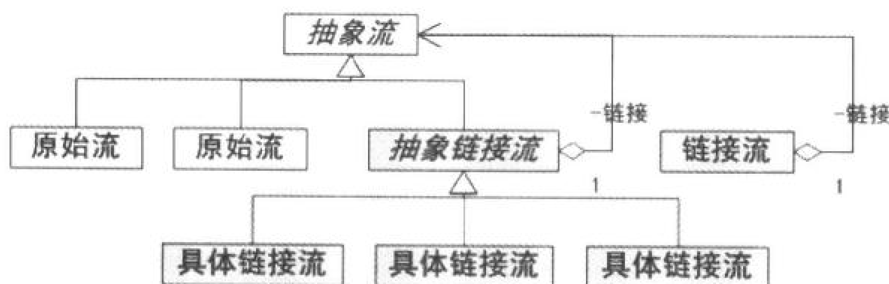
重新并行化。

（3）SeqcueneInputStream 可以将两个已有的输入流连接起来，形成一个输入流，从而将多个输入流排列构成一个输入流序列。

值得指出的是，虽然 PipedInputStream 接收一个流对象 PipedOutputStream 作为流的源，但是，PipedOutputStream 流对象的类型不是 InputStream。因此，PipedInputStream 流处理器仍然属于原始流处理器。

抽象结构图

如果按照上面的这种原始流处理器和链接流处理器的划分，不难看出这些流处理器之间的关系可以用下图所示的结构图来描述。



上面的流处理器图与装饰模式的结构图有着显而易见的相同之处。实际上，InputStream 类型的流处理器结构确实符合装饰模式，而这可以从它们在结构中扮演的角色分辨出来。

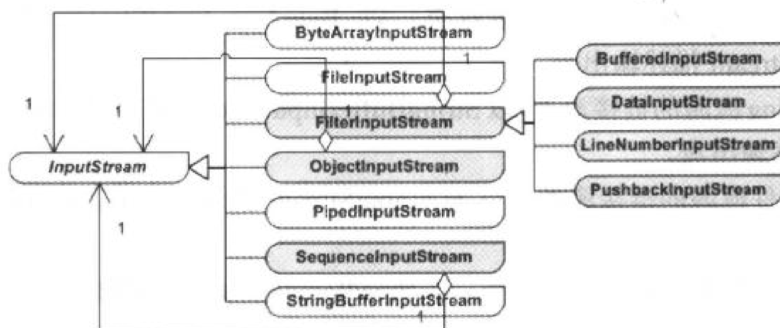
实际上，其他类型的流处理器也都可以抽象化成类似上面的结构图。

装饰模式的各个角色

在所有 InputStream 类型的链接流处理器中，使用频率最大的就是 FilterInputStream 类，以这个类为抽象装饰角色的装饰模式结构非常明显和典型。下面就以这个类为核心说明装饰模式的各个角色是由哪些流处理器扮演的：

- 抽象构件（Component）角色：由 InputStream 扮演。这是一个抽象类，为各种子类型流处理器提供统一的接口。
- 具体构件（Concrete Component）角色：由 ByteArrayInputStream、FileInput Stream、PipedInputStream 以及 StringBufferInputStream 等原始流处理器扮演。它们实现了抽象构件角色所规定的接口，可以被链接流处理器所装饰。
- 抽象装饰（Decorator）角色：由 FilterInputStream 扮演。它实现了 InputStream 所规定的接口。
- 具体装饰（Concrete Decorator）角色：由几个类扮演，分别是 DataInputStream、BufferedInputStream 以及两个不常用到的类 LineNumberInputStream 和 PushbackInputStream。

读者可以看到，所谓的链接流其实就是装饰角色，而原始流就是具体构件角色，如下图所示。



一方面，链接流对象接收一个（同类型的）原始流对象或者另一个（同类型的）链接流对象作为流源；另一方面，它们都对流源对象的内部工作方法做了相应的改变，这种改变是装饰模式所要达到的目的。比如：

（1）BufferedInputStream“装饰”了 InputStream 的内部工作方式，使得流的读入操作使用缓冲机制。在使用了缓冲机制后，不会对每一次的流读入操作都产生一个物理的读盘动作，从而提高了程序的效率。在涉及到物理流的读入时，都应当使用这个装饰流类。

（2）LineNumberInputStream 和 PushbackInputStream 也同样“装饰”了 InputStream 的内部工作方式，前者使得程序能够按照行号读入数据；后者能够使程序在读入的过程中退后一个字符。后两个装饰类可能在实际的编程工作中很少用到，因为它们是为了支持用 Java 语言制作编译器而准备的。

（3）DataInputStream 子类读入各种不同的原始数据类型以及 String 类型的数据，这一点可以从它所提供的各种 read()方法看出来：

- readByte()
- readUnsignedByte()
- readShort()
- readUnsignedShort()
- readChar()
- readInt()
- readLong()
- readFloat()
- readDouble()
- readUTF()

使用这个流处理器以及它的搭档 DataOutputStream，可以将原始数据从一个地方通过流移到另一个地方。

OutputStream 类型中的装饰模式

OutputStream 是一个用于输出的抽象类，它的接口、子类的等级结构、子类的功能都和 InputStream 有很好的对称性。在 OutputStream 给出的接口里，将 write 换成 read 就得到了 InputStream 的接口，而其具体子类则在功能上面是平行的。

（1）针对 byte 数组流源的链接流类，以 ByteArrayInputStream 描述输入流，以 ByteArrayOutputStream 描述输出流。

（2）针对 String 流源的链接流类，以 StringBufferInputStream 描述输入流，以 StringBuffer OutputStream 描述输出流。

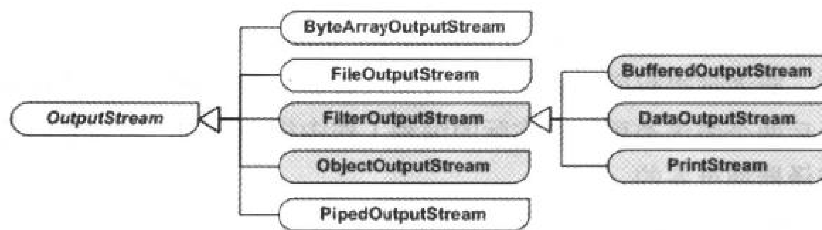
（3）针对文件流源的链接流类，以 FileInputStream 描述输入流，以 FileOutputStream 描述输出流。

（4）针对数据管道流源的链接流类，以 PipedInputStream 描述输入流，以 PipedOutput Stream 描述输出流。

（5）针对由多个流组成的序列，以 SequenceInputStream 描述输入流，以 SequenceOutput Stream 描述输出流。

OutputStream 类型有哪些子类

OutputStream 有五个直接的具体子类，加上三个属于 FilterInputStream 的具体子类，一共有八个具体子类，如下图所示。



原始流处理器

在 OutputStream 类型的流处理器中，原始流处理器包括以下三种：

- ByteArrayOutputSteam：为多线程的通信提供缓冲区操作功能。输出流的汇集是一个 byte 数组。
- FileOutputStream：建立一个与文件有关的输出流。输出流的汇集是一个文件对象。
- PipedOutputStream：可以与 PipedInputStream 配合使用，用于向一个数据管道输出数据。

链接流处理器

OutputStream 类型的链接流处理器包括以下几种：

（1）FilterOutputStream 称为过滤输出流，它将另一个输出流作为流汇。这个类的子类有如下几种。

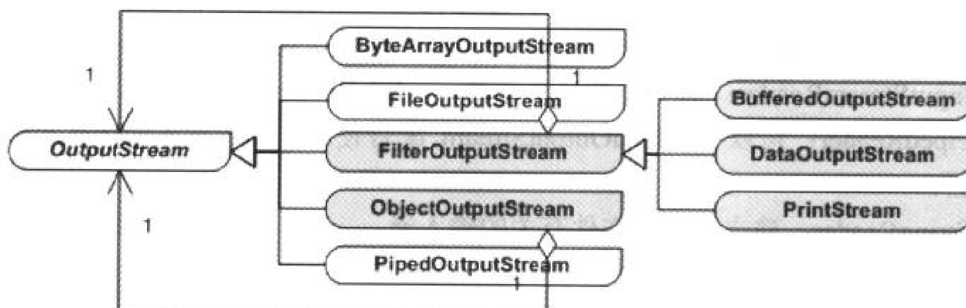
- BufferOutputStream：用来向一个内存缓冲区中写出数据，并将此缓冲区的数据输出到硬盘中。
- DataOutputStream：提供基于多字节的写出方法，可以写出原始数据类型的数据。
- PrintStream：用于产生格式化输出。读者一定会很熟悉 System.out 静态对象，它就是一个 PrintStream。

（2）ObjectOutputStream 可以将原始数据类型和对象串行化。



装饰模式的各个角色

在所有的链接流处理器中，最常见的就是 FilterOutputStream 类。以这个类为核心的装饰模式结构非常明显和典型，如下图所示。



现在来看一看装饰模式所涉及的各个角色是由哪些 Java 的 I/O 类来扮演的：

- 抽象构件（Component）角色：由 OutputStream 扮演。这是一个抽象类，为各种的子类型流处理器提供统一的接口。
- 具体构件（Concrete Component）角色：由 ByteArrayOutputStream、FileOutput Stream 以及 PipedOutputStream 等扮演，它们均实现了 OutputStream 所声明的接口。
- 抽象装饰（Decorator）角色：由 FilterOutputStream 扮演。它有与 OutputStream 相同的接口，而这正是装饰类的关键。
- 具体装饰（Concrete Decorator）角色：由几个类扮演，分别是 BufferedOutput Stream、DataOutputStream，以及 PrintStream。

读者可以再一次看到，所谓的链接流，其实就是装饰模式中的装饰角色，而原始流就是具体构件角色。

与 DataInputStream 相对应的，是 DataOutputStream。后者负责将由原始数据类型和 String 对象组成的数据格式化，并输出到一个流中，使得任何机器上的任何 DataInputStream 类型的对象都可以读入这些数据。所有的方法都是以 write 开始的，比如 writeByte(), writeFloat()等。

如果需要对数据做真正的格式化，以便输出到像控制台显示那样，那就需要使用 PrintStream。PrintStream 可以对由原始数据类型和 String 对象组成的数据进行格式化，以形成可以阅读的格式；而 DataOutputStream 则不同，它将数据输出到一个流中，以便 DataInputStream 可以在任何机器和操作系统中都可以重新将数据读入，并进行结构重建。System.out 就是一个静态的 PrintStream 对象。

PrintStream 对象最为重要的两个方法是 print()和 println()，这两个方法都是重载的，以便可以打印出所有适用类型的数据。这两个方法之间的区别是后者在每一行结束时多打印出一个换行符号。

BufferedOutputStream 对一个输出流进行装饰，使得流的写出操作使用缓冲机制。在使用了缓冲机制之后，不会对每一次的流写入操作都产生一个物理的写动作，从而提高了程序的效率。在涉及到物理流的地方，比如控制台 I/O、文件 I/O 等，都应当使用这个装

饰流处理器。

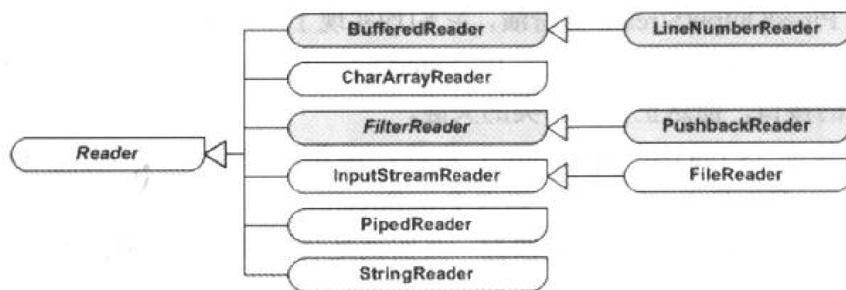
Reader 类型中的装饰模式

实际上，在 Reader 类型的流处理器中，原始流处理器包括以下四种：

- CharArrayReader：为多线程的通信提供缓冲区操作功能。
- InputStreamReader：这个类有一个子类——FileReader。
- PipedReader：可以与 PipedOutputStream 配合使用，用于读入一个数据管道的数据。
- StringReader：建立一个与文件有关的输入流。

而链接流处理器包括以下的几种：

- BufferedReader：用来从硬盘将数据读入到一个内存缓冲区中，并从此缓冲区提供数据，这个类的子类为 LineNumberReader。
- FilterReader：成为过滤输入流，它将另一个输入流作为流的来源。这个类的子类有 PushbackReader，提供基于多字节的读取方法，可以读取原始数据类型的数据，Reader 类型的类图如下图所示。

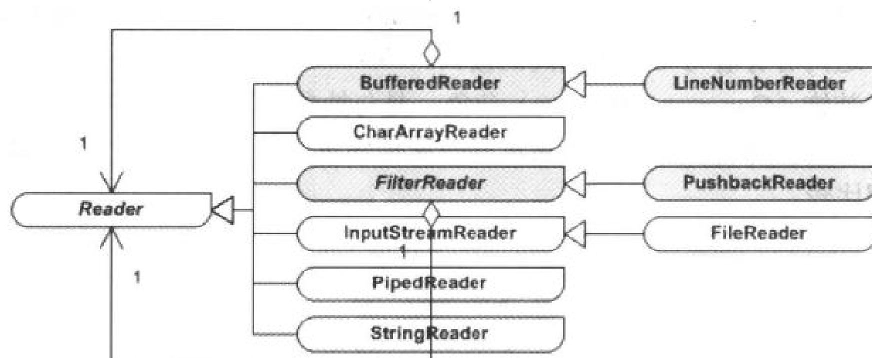


现在来看一看装饰模式所涉及的各个角色是由哪些 Java 的 I/O 类来扮演的：

- 抽象构件（Component）角色：由 Reader 扮演。这是一个抽象类，为各种的子类型流处理器提供统一的接口。
- 具体构件（Concrete Component）角色：由 CharArrayReader、InputStreamReader、PipedReader 以及 StringReader 等扮演，它们均实现了 Reader 所声明的接口。
- 抽象装饰（Decorator）角色：由 BufferedReader 以及 FilterReader 扮演。这两者有着与 Reader 相同的接口，而这正是装饰类的关键。它们分别给出两个装饰角色的等级结构，第一个给出 LineNumberReader 作为具体装饰角色，另一个给出 PushbackReader 作为具体装饰角色。
- 具体装饰（Concrete Decorator）角色：如前所述，分别是 LineNumberReader 作为 BufferedReader 的具体装饰角色，PushbackReader 作为 FilterReader 的具体装饰角色。

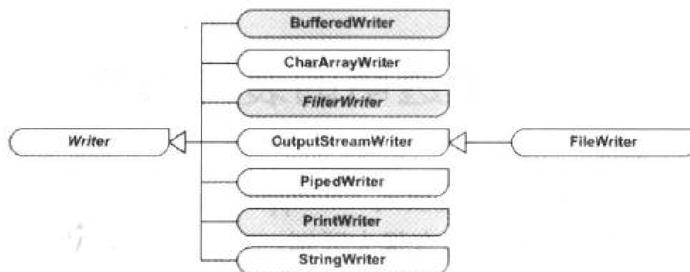
如下图所示，标有聚合联线的就是抽象装饰角色。





Writer 类型中的装饰模式

Writer 类型是一个与 Reader 类型平行的等级结构，而且 Writer 类型的等级结构几乎与 Reader 类型的等级结构关于输入/输出是对称的，如下图所示。



实际上，在 Writer 类型的流处理器中，原始流处理器包括以下四种：

- CharArrayWriter：为多线程的通信提供缓冲区的操作功能。
- OutputStreamWriter：建立一个与文件有关的输出流。含有一个具体子类 FileWriter，为 Writer 类型的输出流提供文件输出功能。
- PipedWriter：可以与 PipedOutputStream 配合使用，用于读入一个数据管道的数据。
- StringWriter：向一个 StringBuffer 写出数据。

而链接流处理器包括以下的三种：

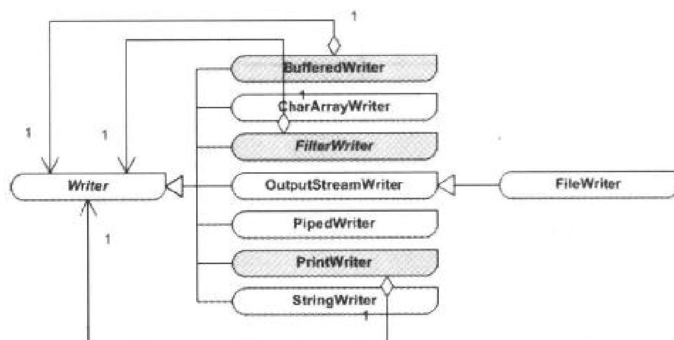
- BufferedWriter：为 Writer 类型的流处理提供缓冲区功能。
- FilterWriter：称为过滤输入流，它将另--个输入流作为流的来源。这是一个没有子类的抽象类。
- PrintWriter：支持格式化的文字输出。

现在来看一看装饰模式所涉及的各个角色是由哪些 Java 的 I/O 类来扮演的：

- 抽象构件（Component）角色：由 Writer 扮演。这是一个抽象类，为各种的子类型流处理器提供统一的接口。
- 具体构件（Concrete Component）角色：由 CharArrayWriter、OutputStream Wrtier、PipedWriter 以及 StringWriter 等扮演，它们均实现了 Reader 所声明的接口。

- 抽象装饰（Decorator）角色：由 BufferedWriter、FilterWriter 以及 PrintWriter 扮演，这两者有着与 Writer 相同的接口。
- 具体装饰（Concrete Decorator）角色：是与抽象装饰角色合并的。

如下图所示，标出了从抽象装饰角色到抽象构件角色的聚合连线，更加易于与装饰模式的结构图相比较。



显然，由于抽象装饰角色与具体装饰角色发生合并，因此装饰模式在这里被简化了。

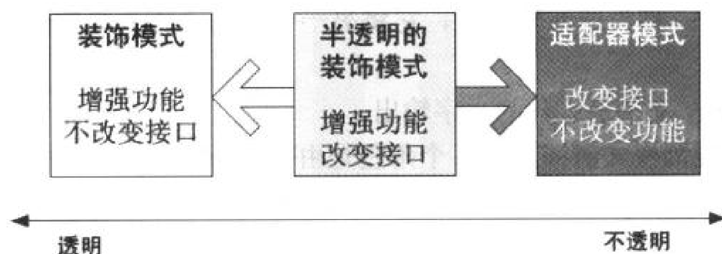
27.4 半透明的装饰模式

装饰模式和适配器模式都是“包裹模式（Wrapper Pattern）”，它们都是通过封装其他对象达到设计目的的，但是它们的形态有很大区别。

理想的装饰模式在对被装饰对象进行功能增强的同时，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。而适配器模式则不然，一般而言，适配器模式并不要求对源对象的功能进行增强，但是会改变源对象的接口，以便和目标接口相符合。

但是，正如本书在“装饰（Decorator）模式”一章中所指出的，装饰模式有透明的和半透明的两种。这两种的区别就在于装饰角色的接口与抽象构件角色的接口是否完全一致。透明的装饰模式也就是理想的装饰模式，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。

相反，如果装饰角色的接口与抽象构件角色接口不一致，也就是说装饰角色的接口比抽象构件角色的接口宽的话，装饰角色实际上已经成了一个适配器角色，这种装饰模式在本书也是可以接受的，称为“半透明”的装饰模式，如下图所示。



在适配器模式里面，适配器类的接口通常会与目标类的接口重叠，但往往并不完全相



同。换言之，适配器类的接口会比被装饰的目标类的接口宽。

显然，半透明的装饰模式实际上就是处在适配器模式与装饰模式之间的灰色地带。如果将装饰模式与适配器模式合并成为一个“包裹模式”的话，那么半透明的装饰模式倒可以成为这种合并后的“包裹模式”的代表。

InputStream 类型中的装饰模式

InputStream 类型中的装饰模式是半透明的。换言之，链接流处理器在对源流处理器（Source Stream Class）进行功能增强的同时，改变了源流处理器的接口。为了说明这一点，不妨看一看作为装饰模式的抽象构件角色的 InputStream 的源代码，如代码清单 1 所示。

代码清单 1：InputStream 的源代码（有删略）

```
package java.io;
public abstract class InputStream {
    public abstract int read() throws I/OException;

    public int read(byte b[]) throws I/OException {...}

    public int read(byte b[], int off, int len) throws I/OException {...}

    public long skip(long n) throws I/OException {...}

    public int available() throws I/OException {...}

    public void close() throws I/OException {...}

    public synchronized void mark(int readlimit) {}

    public synchronized void reset() throws I/OException {...}

    public boolean markSupported()
}
```

可以看出，这个抽象类声明了九个方法，并给出了其中八个的实现，另有一个是抽象方法，需由子类实现。

下面是作为装饰模式的抽象装饰角色的 FilterInputStream 的源代码，如代码清单 2 所示。

代码清单 2：FilterInputStream 的源代码（有删略）

```
package java.io;
public class FilterInputStream extends InputStream {
    protected FilterInputStream(InputStream in) {...}
    public int read() throws I/OException {...}
    public int read(byte b[]) throws I/OException {...}
```

```
        public int read(byte b[], int off, int len) throws I/OException {...}
        public long skip(long n) throws I/OException {...}
        public int available() throws I/OException {...}
        public void close() throws I/OException {...}
        public synchronized void mark(int readlimit) {...}
        public synchronized void reset() throws I/OException {...}
        public boolean markSupported() {...}
    }
```

可以看出，FilterInputStream 的接口与 InputStream 的接口是完全一致的。也就是说，直到这一步，还是与装饰模式相符合的。

这里再查看一下 BufferedInputStream 的源代码，如代码清单 3 所示。

代码清单 3：BufferedInputStream 的源代码（有删略）

```
    package java.io;
    public class BufferedInputStream extends FilterInputStream {
        private void ensureOpen() throws I/OException {...}
        public BufferedInputStream(InputStream in) {...}
        public BufferedInputStream(InputStream in, int size) {...}
        private void fill() throws I/OException {...}
        public synchronized int read() throws I/OException {...}
        private int read1(byte[] b, int off, int len) throws I/OException {...}
        public synchronized int read(byte b[], int off, int len)
        throws I/OException {...}
        public synchronized long skip(long n) throws I/OException {...}
        public synchronized int available() throws I/OException {...}
        public synchronized void mark(int readlimit) {...}
        public synchronized void reset() throws I/OException {...}
        public boolean markSupported() {...}
        public void close() throws I/OException {...}
    }
```

你会发现，这个链接流处理器提供了额外的方法，包括 ensureOpen()、fill()等。

这就意味着 BufferedInputStream 是一个半透明的装饰类。换言之，它是破坏理想的装饰模式的要求的。一个理想的装饰类应当让客户端根本察觉不到所使用的是一个装饰对象，而不是原始流对象。

现在，如果客户端持有一个类型为 InputStream 的对象引用 in 的话，那么如果 in 的真实类型是 BufferedInputStream 的话，只要客户端不需要使用像 ensureOpen()以及 fill()这样的方法，客户端一般没有问题。

但是如果客户端必须使用这两个方法，怎么办呢？当然就必须进行向下类型转换。将 in 的类型转换成为 BufferedInputStream 之后才可能调用这两个方法。但是，这个类型转换意味着客户端必须知道它拿到的引用是指向一个类型为 BufferedInputStream 的对象的。这就破坏了使用装饰的原始用意：不让客户端察觉它所使用的仅是一个装饰，而不是原始流角色。



是的，现实世界与理论总归是有一段差距的。纯粹的装饰模式在真实的系统中很难找到。一般所遇到的，都是这种半透明的（或者称做退化的）装饰模式，也就是处于装饰模式与适配器模式之间的一种灰色地带中的结构。

其他类型中的装饰模式

一个关于装饰模式的重要的事实是，很难找到理想的装饰模式。一般而言，对一个对象进行功能增强，都会导致加入新的行为。因此，装饰角色的接口比抽象构件角色的接口宽是很难避免的。这种现象存在于 Java I/O 库中所有的类型的链接流处理器中。

比如，BufferedReader 是属于 Reader 类型的输入流处理器，这个流处理器的功能是给流的处理增加缓冲功能。既然一个流处理器具有了缓冲功能，那么就可以提供按行（而不是按字）读入的功能，而这就是 BufferedReader 提供的 readLine()方法。

虽然提供这个方法是顺理成章的，但是这个方法是 Reader 接口所没有的（也是不应该有的）。这就使得 BufferedReader 成了半透明的装饰类，或者叫做退化的装饰类。

凡是使用 BufferedReader 的地方，一般都需要使用 readLine()方法。所以，客户端必须声明一个 BufferedReader 类型的流处理器，而不是 Reader 类型的流处理器，才能够调用这个方法。

一个装饰类提供的新的方法越多，它离纯装饰模式的距离也就越远，离适配器模式的距离也就越近。

27.5　适配器模式的应用

适配器模式是 Java I/O 库中第二个最重要的设计模式。

InputStream 原始流处理器中的适配器模式

InputStream 类型的原始流处理器是适配器模式的应用。

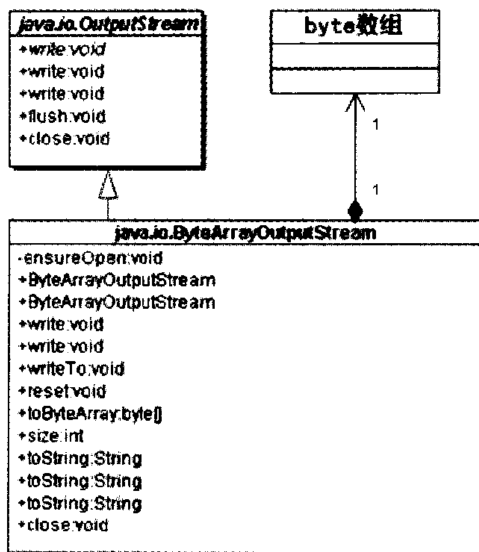
ByteArrayInputStream 是一个适配器类

ByteArrayInputStream 继承了 InputStream 的接口，而封装了一个 byte 数组。换言之，它将一个 byte 数组的接口适配成 InputStream 流处理器的接口。

根据“Java 语言的技术说明书”[JLS00]，Java 语言支持四种类型：Java 接口、Java 类、Java 数组和原始类型。前三种是引用类型（Reference Type），类和数组的实例是对象，原始类型的值不是对象。

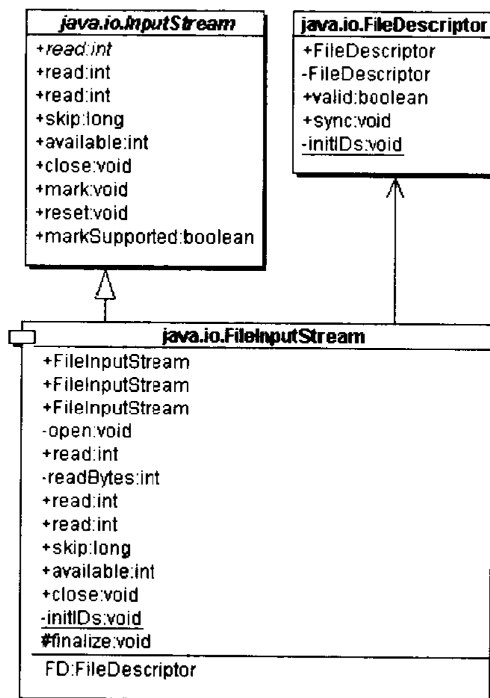
换言之，Java 语言的数组是像所有的其他对象一样的对象，而不管数组中所存储的元素的类型是什么。

这样一来，ByteArrayInputStream 就符合适配器模式的描述，是一个对象形式的适配器类，如下图所示。



FileInputStream 是一个适配器类

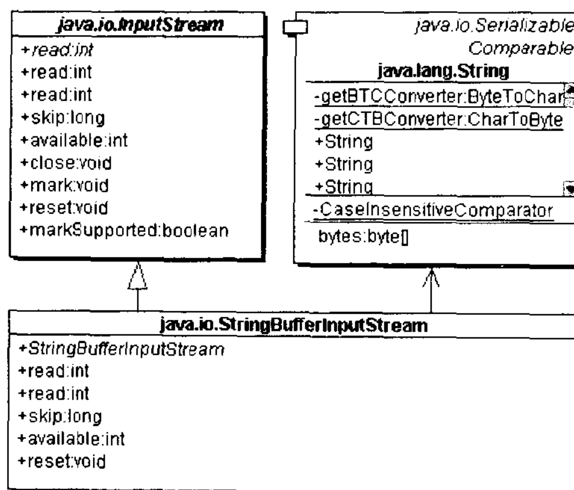
FileInputStream 继承了 InputStream 类型，同时持有一个对 FileDescriptor 的引用。这是一个将 FileDescriptor 对象适配成 InputStream 类型的对象形式的适配器模式，如下图所示。



StringBufferInputStream 是一个适配器类

StringBufferInputStream 继承了 InputStream 类型，同时持有一个对 String 对象的引用。这是一个将 String 对象适配成 InputStream 类型的对象形式的适配器模式，如下图所示。





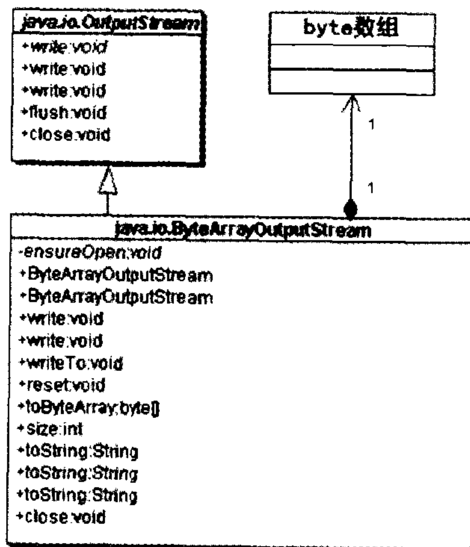
PipedInputStream 是与 PipedOutputStream 一起存在的，因此，关于这个流处理器的分析推迟到讨论 PipedOutputStream 时。

OutputStream 原始流处理器中的适配器模式

同样，在 OutputStream 类型中，所有的原始流处理器都是适配器类。

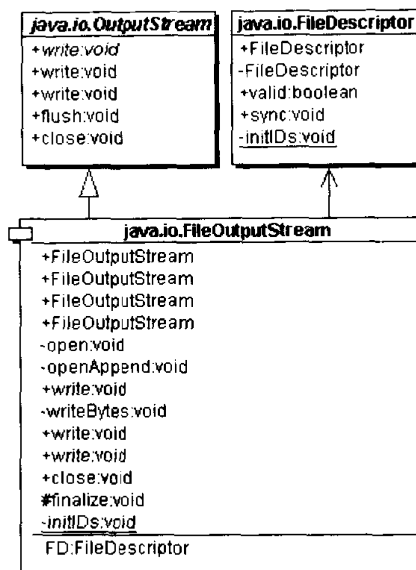
ByteArrayOutputStream 是一个适配器类

ByteArrayOutputStream 继承了 OutputStream 类型，同时持有一个对 byte 数组的引用。它把一个 byte 数组的接口适配成 OutputStream 类型的接口，因此也是一个对象形式的适配器模式的应用，如下图所示。



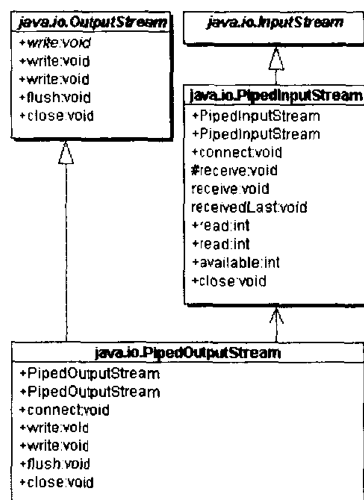
FileOutputStream 是一个适配器类

FileOutputStream 继承了 OutputStream 类型，同时持有一个对 FileDescriptor 对象的引用。这是一个将 FileDescriptor 接口适配成 OutputStream 接口的对象形式的适配器模式，如下图所示。



PipedOutputStream 是一个适配器类

PipedOutputStream 总是与 PipedInputStream 一同使用的。它接收一个类型为 PipedInputStream 的输入类型，并将之转换成类型为 OutputStream 类型的输出流。这是一个对象形式的适配器模式的应用，如下图所示。

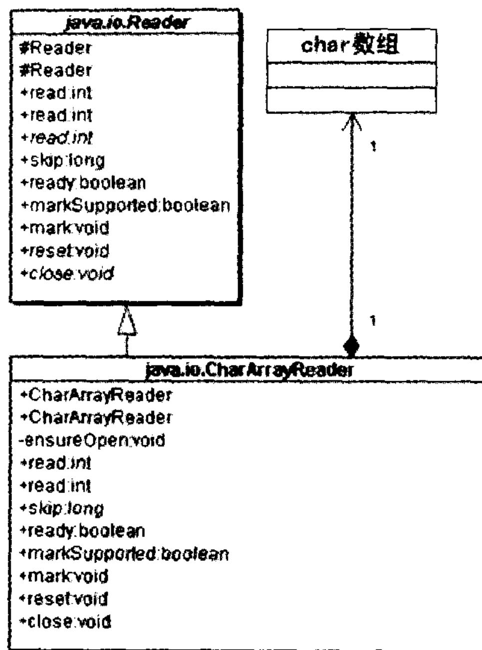


Reader 原始流处理器中的适配器模式

Reader 类型的原始流处理器都是适配器模式的应用。

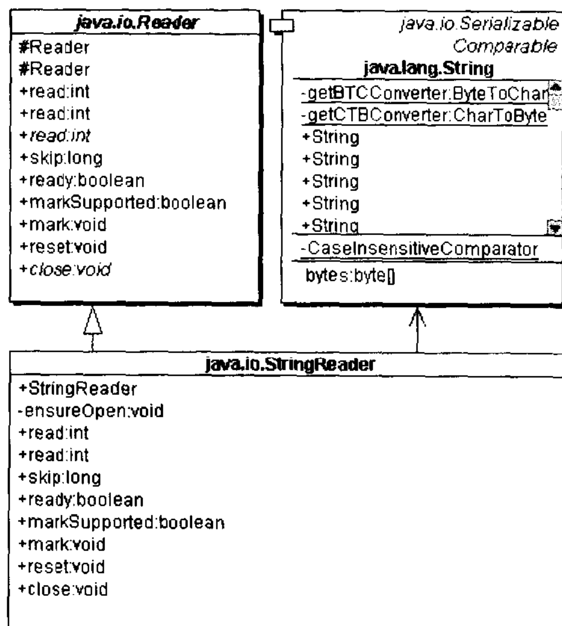
CharArrayReader 是一个适配器类

CharArrayReader 将一个 char 数组适配成为 Reader 类型的输入流，因此，它是一个对象形式的适配器模式的应用，如下图所示。



StringReader 是一个适配器类

StringReader 继承了 Reader 类型，持有一个对 String 对象的引用。它将 String 的接口适配成 Reader 类型的接口，如下图所示。



InputStreamReader 是从 byte 流到 char 流的适配器，关于 InputStreamReader 会在下面做专题讨论。

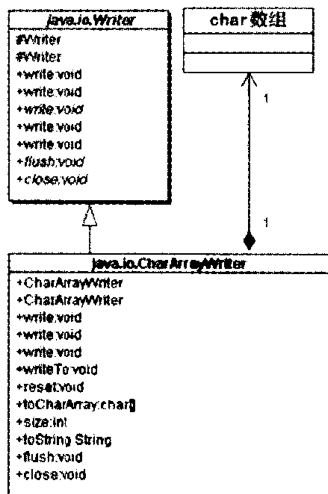
PipedReader 是与 PipedWriter 一起存在的，因此，关于这个流处理器的分析推迟到下面讨论 PipedWriter 时。

Writer 类型中的适配器模式

Writer 类型的原始流处理器是适配器模式的具体应用。

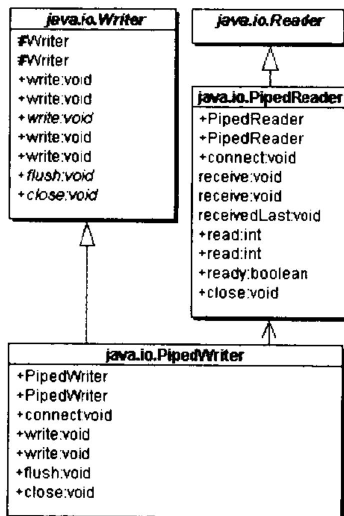
CharArrayWriter 是一个适配器类

CharArrayWriter 将一个 char 数组适配成为 Writer 接口，如下图所示。



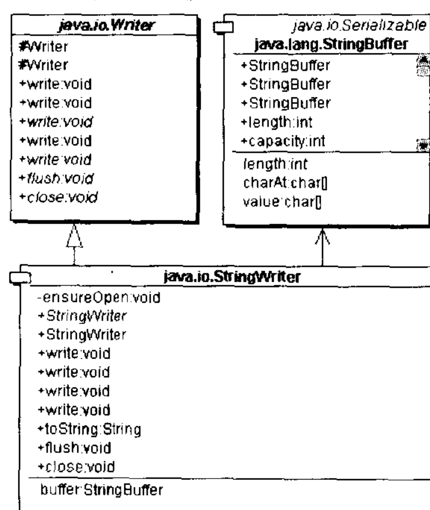
PipedWriter 是一个适配器类

PipedWriter 总是与 PipedReader 一同使用的。它将一个 PipedReader 对象的接口适配成一个 Writer 类型的接口，如下图所示。



StringWriter 是一个适配器类

StringWriter 继承了 Writer 类型，同时持有一个 StringBuffer 对象。它将 StringBuffer 对象的接口适配成为了 Writer 类型的接口，是一个对象形式的适配器模式的应用，如下图所示。



27.6　从 byte 流到 char 流的适配

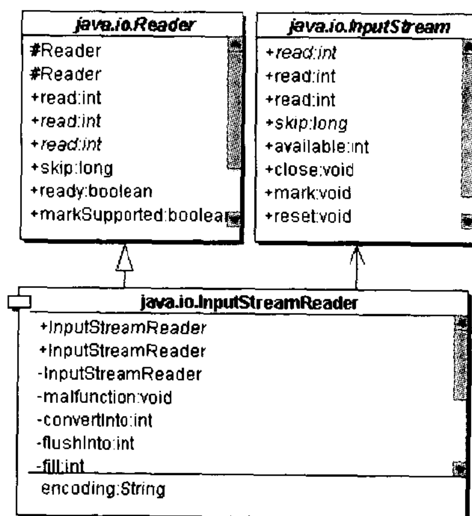
在 Java 语言的标准库 Java I/O 里面，有一个 InputStreamReader 类叫做桥梁（bridge）类。在 JDK 1.3 的文档里针对这个类介绍说：

"InputStreamReader 是从 byte 流到 char 流的一个桥梁，它读入 byte 数据并根据指定的编码将之翻译成为 char 数据。"

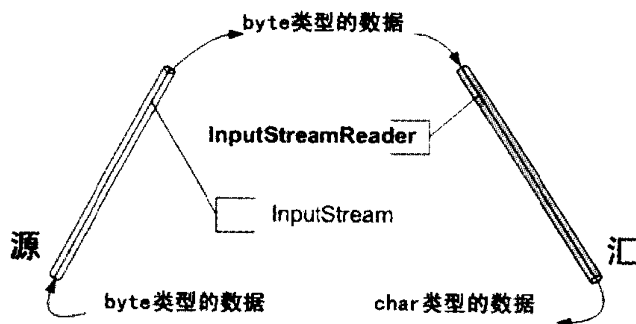
InputStreamReader 虽然叫做"桥梁"，但是它不是桥梁模式的应用，而是适配器模式的应用。

InputStreamReader

InputStreamReader 是从 byte 输入流到 char 输入流的一个适配器。下图所示就是 InputStreamReader 与 Reader 和 InputStream 等类的结构图。

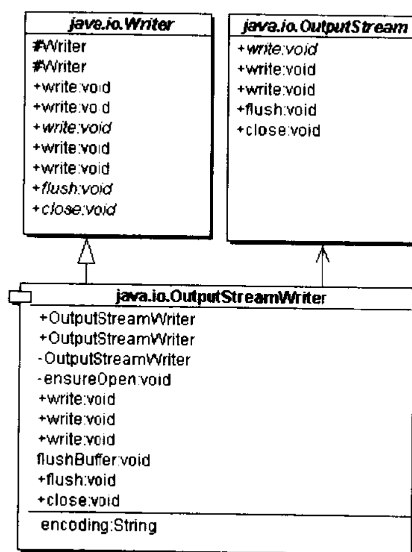


当把 InputStreamReader 与任何 InputStream 的具体子类链接时，可以从 InputStream 的输出读入 byte 类型的数据，将之转换成为 char 类型的数据，如下图所示。

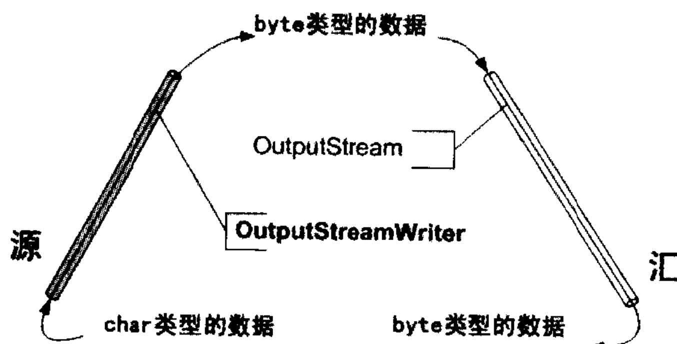


OutputStreamWriter

同样地，InputStreamReader 是从 byte 输出流到 char 输出流的一个适配器。或者说，OutputStreamWriter 是从 OutputStream 到 Writer 的适配器类，如下图所示。



换言之，当与任何一个 OutputStream 的具体子类相链接时，OutputStreamWriter 可以将 char 类型的数据转换成为 byte 类型的数据，再交给输出流，如下图所示。





一个例子

为了说明适配器类 InputStreamReader 是如何使用的，下面特别给出一个例子。Echo 类可以将控制台输入的任何字符（包括中文和英文字符）重新打印出来，就像回声一样。

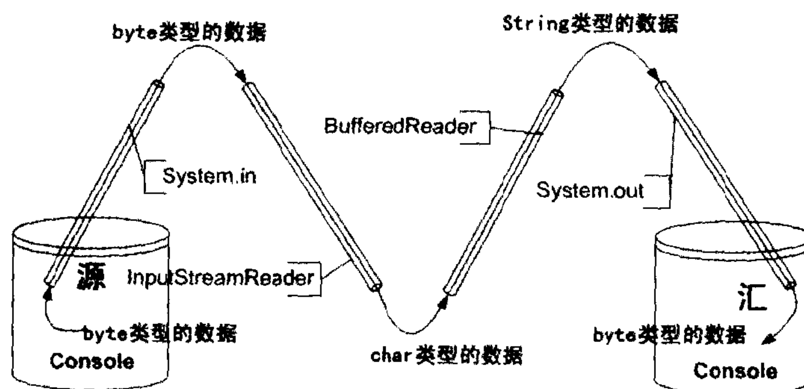
这个类的源代码如代码清单 4 所示。

代码清单 4：Echo 的源代码

```
package com.javapatterns.javaio.adapter;
import java.io.*;
import java.io.BufferedReader;
public class Echo
{
    public static void main(String[] args) throws IOException
    {
        String line;
        InputStreamReader input = new InputStreamReader(System.in);
        System.out.println("Enter data and push enter: ");
        BufferedReader reader = new BufferedReader(input);
        line = reader.readLine();
        System.out.println("Data entered: " + line);
    }
}
```

可以看出，这个类接收一个类型为 InputStream 的 System.in 对象，将之适配成为 Reader 类型；然后再使用 BufferedReader 类“装饰”它，将缓冲功能加上去。这样一来，就可以使用 BufferedReader 对象的 readLine()方法读入整行的输入数据，数据类型是 String。

在得到这个数据之后，程序又将它写出到 System.out 中去，完成了全部的流操作。下图所示为其管道图。



值得指出的是，本系统使用了 BufferedReader 来为流的读入提供缓冲功能。这样做的直接效果是可以使用 readLine()方法按行读入数据。但是由于 Reader 接口并不提供 readLine()方法，所以这样一来，系统就必须声明一个 BufferedReader 类型的流处理器，而不是一个 Reader 类型的流处理器。根据本章前面的讨论，这意味着装饰模式的退化。

在上面的管道链接过程当中，InputStreamReader 起到了适配器的作用，它将一个 byte 类型的输入流适配成为一个 char 类型的输入流。在这之后，BufferedReader 则起到了装饰模式的作用，将缓冲机制引入到流的读入中。

因此，这个简单的例子涉及了两个设计模式。

中文信息的读写

Java 语言的 I/O 库提供了两套平行独立的等级结构，即 InputStream、OutputStream 和 Reader、Writer。前者处理 8 位元的流，后者处理 16 位元的流。由于 Java 语言的 byte 数据类型是 8 位元的，而 char 是 16 位元的，所以中文信息需要用第二个等级结构的类才能读写。

几乎所有的 8 位元的 Java 语言的 I/O 库类都有相应的 16 位元的 I/O 库类。但是，有很多时候需要把两种库类结合起来用。这时候，Java 语言所提供的桥梁类就变得很有用。例如，InputStreamReader 把 InputStream 适配到 Reader，而 OutputStreamWriter 把 OutputStream 适配到 Writer 类。换言之，它们把 byte 流适配成为 char 流。

例如，因为英文的 DOS 窗口的字集是（ISO Latin-1），如果要从英文的 DOS 窗口读入中文输入，就需要把 System.in 适配到 Reader：

```
InputStreamReader isr = new InputStreamReader(System.in);
```

另一方面，如果要写到 DOS 窗口去，就需要把 Writer 适配成 System.out：

```
OutputStreamWriter osw = new OutputStreamWriter(System.out);
```

如果要读取一个在 Macintosh 写好的文件，需要这样：

```
FileInputStream fis = new FileInputStream("symbol.txt");
InputStreamReader isr = new InputStreamReader(fis, "MacSymbol");
```

反过来，向这个文件里写则需要：

```
FileOutputStream fos = new FileOutputStream("symbol.txt");
OutptuStreamWriter osw = new OutputStreamWriter(fos, "MacSymbol");
```

问答题

1. 在装饰模式中，抽象构件角色的接口是否有可能比具体装饰角色的接口宽？
2. 在适配器模式中，适配器角色的接口是否有可能比目标接口窄？
3. 如前所述，数组实际上是 Java 语言类型中的“一等公民”。那么能不能像将一个普通类实例化一样，将一个数组实例化呢？
4. 请使用本书在“装饰（Decorator）模式”一章中引入的包裹图来描述 InputStream Reader 的工作方式。

问答题答案

1．这是不可能的。抽象构件角色是具体装饰角色的超类型，根据里氏代换原则，任何使用超类型的地方都可以使用子类型。这就是说，子类型一定具有超类型的接口，不然就会违反里氏代换原则。

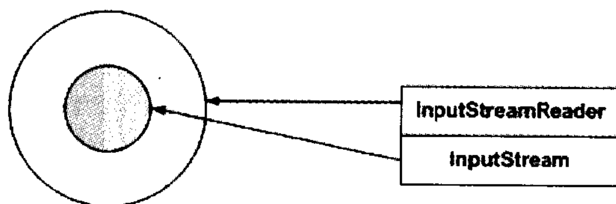
Java 编译器强制所有的子类型都实现超类型的接口，不然就会给出编译时期错误。换言之，超类型的接口只可能比子类型的窄，而不可能比子类型的宽。

2．这是不可能的。适配器角色是目标角色的子类型。而根据里氏代换原则，适配器角色必须实现目标角色的所有接口。如果不满足这个条件，Java 语言的编译器会给出编译时期错误。

3．数组可以像普通类一样使用关键字 new 实例化。下面就将一个元素类型为 int 原始类型的数组实例化：

```
int[] x = new int[5];
```

4．流处理器 InputStreamReader 接收一个 InputStream 类型的流源，提供一个 Reader 类型的接口。下图所示就是描述这个处理器工作方式的包裹图。



参考文献

[DAI99]戴梅萼等编著．Java 问答式教程．清华大学出版社，1999

[HAROLD99] Elliotte Rusty Harold．Java I/O．published by O'Reilly，March 1999

[LISKOV01] Barbara Liskov with John Guttag．Program Development in Java – Abstraction，Specification，and Object-Orientd Design．Addison-Wesley，2001

[JLS00] James Gosling，Bill Joy，Guy Steele，Gilad Bracha．The Java Language Specification．Second Edition，Addison-Wesley，Boston，2000

[BLOCH01] Joshua Bloch．Effective Java – Programming Language Guide．published by Addison-Wesley，2001

[SYBEX99] Matthew Fielder，Kirky Ringer，and John Zukowski (eds.)．Java 2 Complete．published by Sybex，1999

第 28 章　代理（Proxy）模式

代理模式是对象的结构模式[GOF95]。代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。

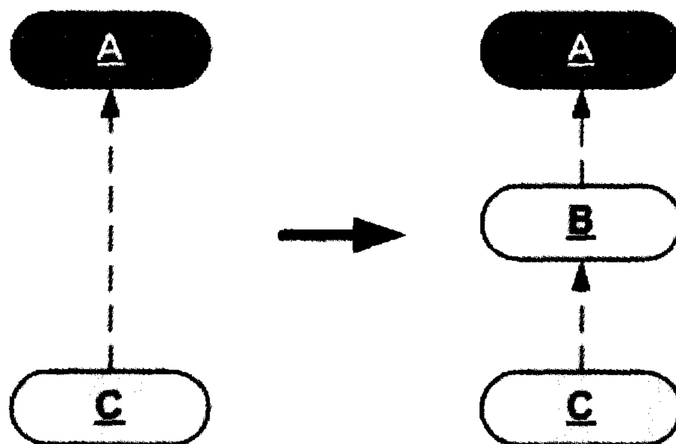
28.1　引　　言

间接的“美”

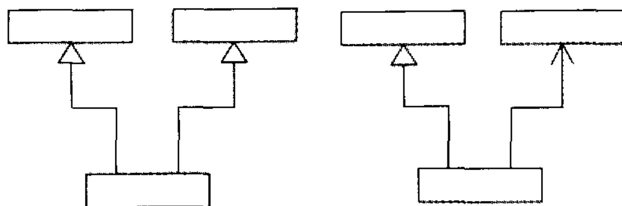
中国人是一个含蓄的民族，讲求微妙和间接的交流方式。对象间的间接通信也同样是面向对象的设计中一条重要的“审美观”。间接性的通信可以给出较低的耦合关系，较强的合作关系，以及微妙的结构和易于复用的设计架构。

假设一个系统已经有一个对象 A。现在需要向这个系统加入一个新的对象 C。对象 A 需要使用系统中已有的对象 A，但是 C 不能像系统现在使用 A 这样使用 A，要么是 A 提供的接口不符合 C 的要求，要么是 A 的功能需要进一步加强等。

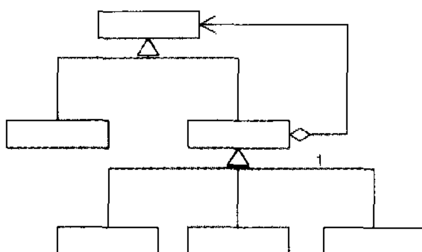
这时候设计师可以选择修改对象 A，也可以选择使用一个中介对象 B，让对象 B 将调用传递给对象 A。这样新的对象 C 便不必直接与系统中已有的对象 A 打交道，而是通过对象 B 与 A 打交道；而对象 B 则可以利用这一有利位置为对象 C 提供一个与 A 完全不同的接口，或者做一些 A 本来不做的操作。如下图所示，左边是 C 与 A 直接通信的示意图，右边是 C 通过中介对象 B 与 A 通信的示意图。



熟悉适配器模式的读者可以立即看出，适配器模式就是这种通信间接化思想的体现，如下图所示，左边是类的适配器模式，右边是对象的适配器模式。



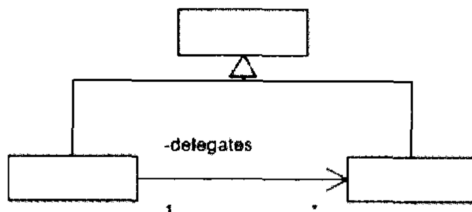
此外，装饰模式也是一个利用了中间对象来增强对象功能的模式，如下图所示。



但是适配器模式和装饰模式并不是惟一强调通信间接化的模式。本章将要引进的是另外一个利用间接通信改善系统设计的模式——代理模式。

代理模式

代理模式的英文叫做 Proxy 或 Surrogate，中文都可译成“代理”。所谓代理，就是一个人或者一个机构代表另一个人或者另一个机构采取行动。 在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。代理模式的简略类图如下图所示。



28.2 代理的种类

如果按照使用目的来划分，代理有以下几种：

- 远程（Remote）代理：为一个位于不同的地址空间的对象提供一个局域代表对象。这个不同的地址空间可以是在本机器中，也可是在另一台机器中。远程代理又叫做大使（Ambassador）。
- 虚拟（Virtual）代理：根据需要创建一个资源消耗较大的对象，使得此对象只在需要时才会被真正创建。本章下面给出一个加载图像的例子说明虚拟代理的使用。



- Copy-on-Write 代理：虚拟代理的一种。把复制（克隆）拖延到只有在客户端需要时，才真正采取行动。
- 保护（Protect or Access）代理：控制对一个对象的访问，如果需要，可以给不同的用户提供不同级别的使用权限。
- Cache 代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
- 防火墙（Firewall）代理：保护目标，不让恶意用户接近。
- 同步化（Synchronization）代理：使几个用户能够同时使用一个对象而没有冲突。
- 智能引用（Smart Reference）代理：当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来等。

在所有种类的代理模式中，虚拟（Virtual）代理、远程（Remote）代理、智能引用代理（Smart Reference Proxy）和保护（Protect or Access）代理是最为常见的代理模式。

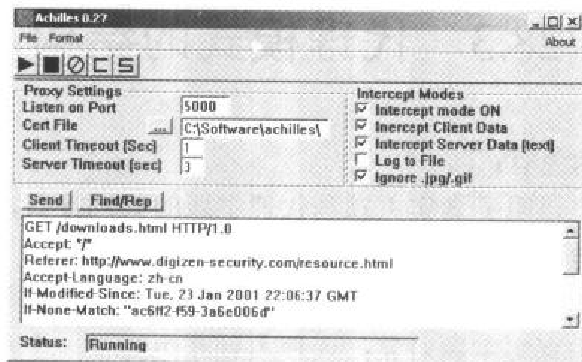
在有些讲解设计模式的书籍中（如[GRAND98]），不同的代理模式被独立划分出来，以强调它们的不同。在另外的书籍中，所有的代理模式都放在一些讲解中，以强调它们的共同点。本书打算采取一种不同的讲解方式，首先将所有的代理模式放到一个地方进行理论上的讲解，然后针对不同的类型提供不同的例子，进行实现方式上的讲解。

远程代理的例子：Achilles

Achilles 是一个用来测试网站的安全性能的工具软件。Achilles 相当于位于客户端的一个桌面代理服务器，在一个 HTTP 过程里起到一个中间人的作用，但是 Achilles 与通常的代理服务器又有不同。

一个通常的 HTTP 代理软件会将一个客户端的 HTTP 数据包转发给网络服务器。Achilles 则截获双向的通信数据，使得 Achilles 软件的用户可以改变来自和发往网络服务器的数据。比如，在一个正常的 SSL 联系中，一个通常的代理服务器会转发通信使得双方可以商议 SSL 连接，而 Achilles 则不同。当 Achilles 处于截取状态时，它会向客户端假装是服务器，同时向真正的服务器假装是浏览器，在两端商议 SSL 通信。Achilles 可以破解加密的数据，给 Achilles 的用户显示已经解密的内容，并且允许用户更改处于通信过程中的数据。

Achilles 软件运行时的情况如下图所示。



Achilles 软件可以免费从 http://www.digizen-security.com/projects.html 下载。

显然，对于浏览器而言，Achilles 所代理的是远程的网络服务器。Achilles 的工作方式便是远程代理模式的应用。

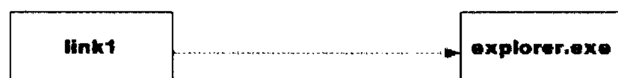
Windows 的快捷方式：代理的例子



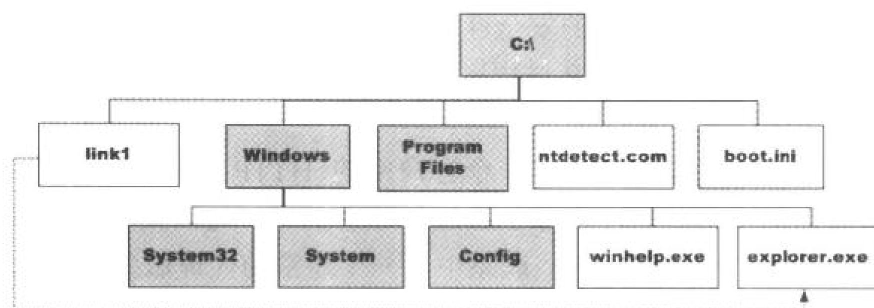
Windows 系统提供快捷方式（Shortcut），可以使任何对象同时出现在多个地方而不必修改原对象。对快捷方式的调用完全与对原对象的调用一样，换言之，快捷方式对客户端是完全透明的，如右图所示。

比如上面的图标便是 Windows 服务的代理。所有的快捷方式都有一个小的箭头在图标的左下方。这就是说，用户可以区分原对象和指向原对象的代理。如果原对象被删除，则快捷方式虽然仍可存在，但是在调用时会给出错误。

如下图所示，一个名为 link1 的快捷方式是一个名为 explorer.exe 的文件的代理。当用户启动这个快捷方式时，link1 会把用户的调用传递给它所代理的 explorer.exe 文件。



如果将这个快捷方式放到文件系统 C:\中，那么它们的关系和位置将如下图所示。



读者可以把它与合成模式中的例子做一个比较。

在 Macintosh 里面有 Alias，在 UNIX 里面有 link，它们都和 Windows 的快捷方式一样，是代理模式的应用。

28.3 代理模式的结构

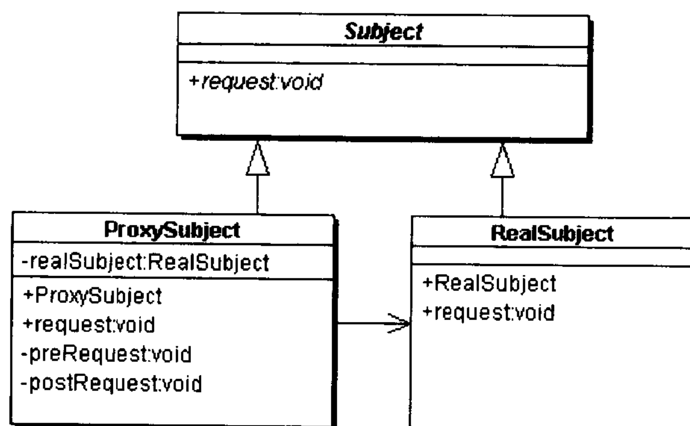
代理模式所涉及的角色有：

- 抽象主题角色：声明了真实主题和代理主题的共同接口，这样一来在任何可以使用真实主题的地方都可以使用代理主题。
- 代理主题（Proxy）角色：代理主题角色内部含有对真实主题的引用，从而可以在任何时候操作真实主题对象；代理主题角色提供一个与真实主题角色相同的接

口，以便可以在任何时候都可以替代真实主体；控制对真实主题的引用，负责在需要的时候创建真实主题对象（和删除真实主题对象）；代理角色通常在将客户端调用传递给真实的主题之前或者之后，都要执行某个操作，而不是单纯地将调用传递给真实主题对象。

- 真实主题角色：定义了代理角色所代表的真实对象。

这里给出一个非常简单的示意性实现，实现的类图如下图所示。



下面给出的是抽象主题角色的示意性源代码，如代码清单 1 所示，可以看出，这个角色规定所有的主题对象必须实现 request()方法。

代码清单 1：抽象主题角色的源代码

```
package com.javapatterns.proxy;
abstract public class Subject
{
    /**
     *  声明一个抽象的请求方法
     */
    abstract public void request();
}
```

下面就是具体主题角色的示意性源代码，如代码清单 2 所示。这里仅仅给出了 request()方法的示意性实现。

代码清单 2：真实主题角色的源代码

```
package com.javapatterns.proxy;
public class RealSubject extends Subject
{
    /**
     *  构造子
     */
    public RealSubject()
    {
    }
```

```
    /**
     *    实现请求方法
     */
    public void request()
    {
        System.out.println("From real subject.");
    }
}
```

下面是代理主题角色的源代码，如代码清单 3 所示。可以看出，代理主题除了将所有的请求原封不动地委派给真实主题角色之外，还在委派之前和之后分别执行一个 preRequest()操作和一个 postRequest()操作。

代码清单 3：代理主题角色的源代码

```
package com.javapatterns.proxy;
public class ProxySubject extends Subject
{
    private RealSubject realSubject;

    /**
     *    构造子
     */
    public ProxySubject()
    {
    }

    /**
     *    实现请求方法
     */
    public void request()
    {
        preRequest();
        if( realSubject == null )
        {
            realSubject = new RealSubject();
        }
        realSubject.request();
        postRequest();
    }
    /**
     *    请求前的操作
     */
    private void preRequest()
    {
        //something you want to do before requesting
```

```
    }
    /**
     *   请求后的操作
     */
    private void postRequest()
    {
        //something you want to do after requesting
    }
}
```

在使用代理主题时，要将变量的明显类型声明为抽象主题的类型，而将其真实类型设置为代理主题类型，如代码清单 4 所示。

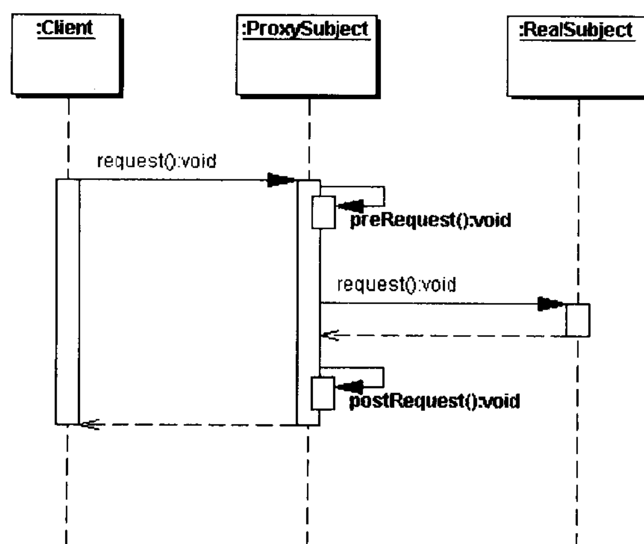
代码清单 4：怎样调用代理主题

```
Subject subject = new ProxySubject();
subject.request();
```

从上面的代理主题类的示意性源代码可以看出代理模式是怎样工作的。首先，代理主题并不改变主题的接口，因为模式的用意是不让客户端感觉到代理的存在；其次，代理使用委派将客户端的调用委派给真实的主题对象，换言之，代理主题起到的是一个传递请求的作用；最后，代理主题在传递请求之前和之后都可以执行特定的操作，而不是单纯传递请求。

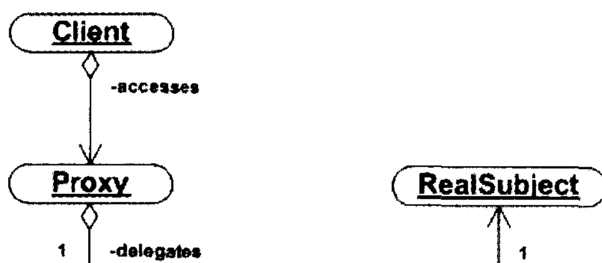
28.4　代理模式的时序

静态的类图并不适合于反映出模式在运行时的特性，时序图更能够反映出模式的活动情况。下图所示就是所讨论的代理模式的时序图。

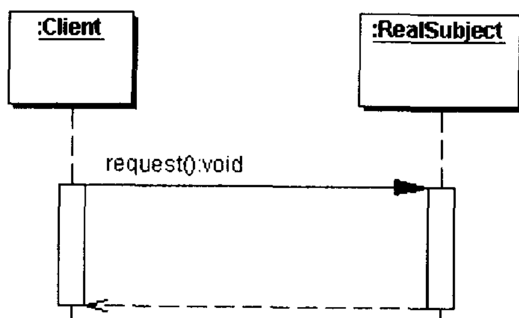


从上面的时序图可以看出，客户端向代理主题发出请求，代理主题在接到请求的同时，

执行了一个 preRequest()操作，然后才把请求传递给真实主题。在真实主题将请求返回后，代理主题又执行了一个 postRequest()操作，才将控制返回给客户端。代理模式的对象图如下图所示。



如下图所示，与客户端直接向真实主题发出请求的情况相比，使用代理主题的显著好处是，系统提供了向真实主题传递客户端请求的控制。代理主题可以在向真实主题传递客户端请求之前执行特定的操作，并决定是否将请求传递给真实主题；代理主题可以在向真实主题传递客户端请求之后执行另外一种操作，比如将客户端请求计数等。客户端有可能根本没有直接向真实主题提出请求的许可，而代理主题具有这种许可，因此，代理主题可以在执行 preRequest()操作后决定是否再传递请求。总之，代理模式将一个中间层插入到客户端和主题角色之间，从而提供了许多的灵活性。

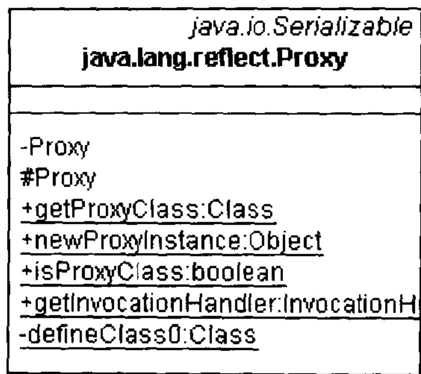


28.5 Java 2.0 对代理模式的支持

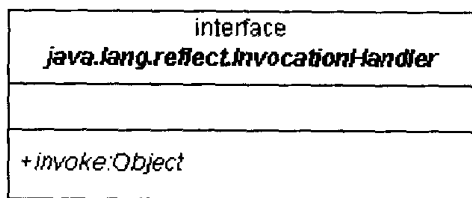
反身映射（Reflection）与动态代理

自从 JDK 1.3 以来，Java 语言通过在 java.lang.reflect 库中提供下面三个类直接支持代理模式：Proxy，InvocationHandler 和 Method。

其中 Proxy 类使得设计师能够在运行时间创建代理对象，其类图如下图所示。



当系统有了一个代理对象后，对原对象的方法调用会首先被分派给一个调用处理器（Invocation Handler）。InvocationHandler 接口的类图如下图所示。



程序可以在调用处理器的 invoke()方法中截获这个调用，进行额外的操作。显然，Java 所提供的这一支持是建立在反身映射（Reflection）的基础之上的。

换言之，设计师可以按照下面的步骤创建动态代理对象：

（1）指明一系列的接口来创建一个代理对象。

（2）创建一个调用处理器（Invocation Handler）对象。

（3）将这个代理指定为某个其他对象的代理对象。

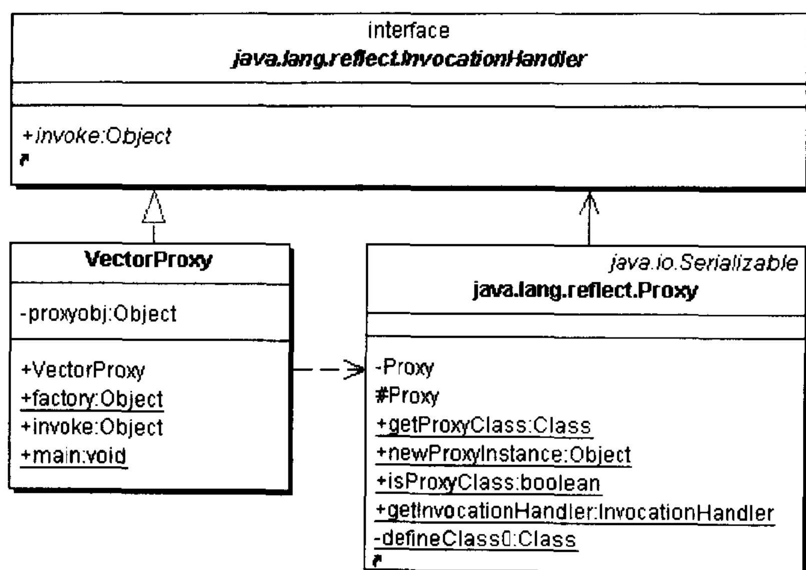
（4）在调用处理器的 invoke()方法中采取代理，一方面将调用传递给真实对象，另一方面执行各种需要做的操作。

一个例子

百闻不如一见，这里提供一个具体的例子讲解 java.lang.reflect.Proxy 的使用方法。首先，这个例子要解决的问题是为一个 Vector 对象提供一个代理对象。当 Vector 的任何方法被调用之前和调用之后，分别打印出两条信息，这表明代理对象有能力截获和控制这个 Vector 对象。

这是一个极其简单的代理例子，但是它具备代理模式所要求的功能。在实际系统中，设计师可以使用这个例子作为蓝本，在一个 Vector 对象（或者任何其他对象）被调用之前和之后做出有意义的操作。

系统仅仅需要一个新的类，称为 VectorProxy 类。此类需要实现 java.lang.reflect.InvocationHandler 接口，也就是说需要实现 invoke()方法，如下图所示。



可以看出，VectorProxy 类必须实现 invoke()方法，因为这是 InvocationHandler 接口的要求。由于 Vector 对象实现了 java.util.List 接口，因此，通过调用 List 的方法，可以操控代理对象。其源代码如代码清单 5 所示。

代码清单 5：VectorProxy 类的源代码

```
package com.javapatterns.proxy.reflect;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
import java.lang.reflect.Method;
import java.util.Vector;
import java.util.List;
public class VectorProxy
    implements InvocationHandler
{
    private Object proxyobj;
    /**
     *  构造子
     */
    public VectorProxy(Object obj)
    {
        proxyobj = obj;
    }
    /**
     *  静态工厂方法
     */
    public static Object factory(Object obj)
    {
        Class cls = obj.getClass();
        return
            cls.getClassLoader(),
```

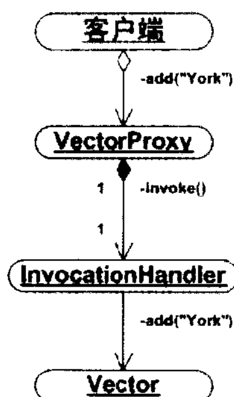
```
            cls.getInterfaces(),
            new VectorProxy(obj) );
    }
    /**
     *    调用某个方法
     */
    public Object invoke(Object proxy,
        Method method, Object[] args)
        throws Throwable
    {
        System.out.println("before calling "
            + method);
        if (args != null)
        {
            for (int i=0; i < args.length; i++)
            {
                System.out.println(args[i] + "");
            }
        }
        Object o = method.invoke(proxyobj, args);
        System.out.println("after calling "
            + method);
        return o;
    }
    public static void main(String[] args)
    {
        List v = null;
        v = (List) factory(new Vector(10));
        v.add("New");
        v.add("York");
    }
}
```

在运行时，系统打印出的结果如代码清单 6 所示。

代码清单 6：运行的结果

```
before calling public abstract boolean
    java.util.List.add(java.lang.Object)
New
after calling public abstract boolean
    java.util.List.add(java.lang.Object)
before calling public abstract boolean
    java.util.List.add(java.lang.Object)
York
after calling public abstract boolean
    java.util.List.add(java.lang.Object)
```

也就是说，代理对象截获了对 Vector 对象的所有调用。在将调用传递给 Vector 之前和之后，代理对象具有采取合适操作的灵活性，虽然这里代理对象所采取的操作不过就是打印出两种信息。本例子的对象图如下图所示。



简单工厂模式

在 VectorProxy 对象中使用了一个静态工厂方法 factory()，负责创建出 Proxy 类的实例，它将创建实例的细节隐藏起来。熟悉简单工厂模式的读者可以辨认出，这是一个简单工厂模式的应用。

28.6 高老庄悟空降八戒

尽管那时候八戒还不叫八戒，但为了方便，这里仍然这样称呼他。

高老庄的故事

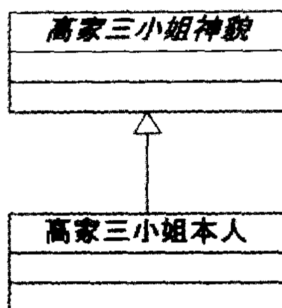
却说那春融时节，悟空牵着白马，与唐僧赶路西行。忽一日天色将晚，远远地望见一村人，这就是高老庄，猪八戒的丈人高太公家。为了将高家三小姐解救出八戒的魔掌，悟空决定扮做高小姐，会一会这个妖怪：

“行者却弄神通，摇身一变，变得就如那女子一般，独自个坐在房里等那妖精。不多时，一阵风来，真个是走石飞砂……那阵狂风过处，只见半空里来了一个妖精，果然生得丑陋：黑脸短毛，长喙大耳，穿一领青不青、蓝不蓝的梭布直裰，系一条花布手巾……走进房，一把搂住，就要亲嘴……”

高三小姐的神貌和本人

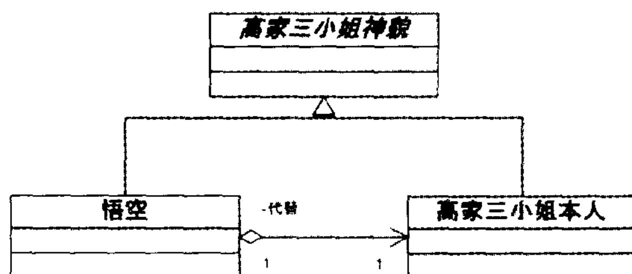
悟空的下手之处是将高三小姐的神貌和她本人分割开来，这和“开–闭”原则有异曲同工之妙。这样一来，“高三小姐本人”也就变成了“高三小姐神貌”的具体实现，而“高三小姐神貌”则变成了抽象角色，如下图所示。





悟空扮演并代替高三小姐

悟空巧妙地实现了“高三小姐神貌”，也就是说同样变成了“高三小姐神貌”的子类。悟空可以扮演高三小姐，并代替高三小姐会见八戒，其静态结构图如下图所示。



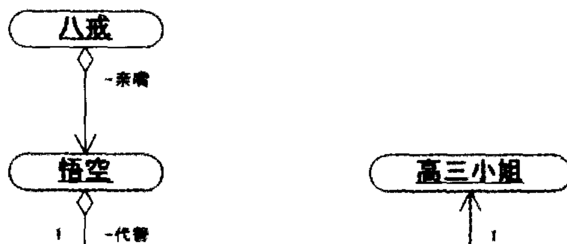
悟空代替“高三小姐本人”去会见猪八戒。

显然这就是代理模式的应用。具体地讲，这是保护代理模式的应用。只有代理对象认为合适时，才会将客户端的请求传递给真实主题对象。

八戒分辨不出真假老婆

从《西游记》的描述可以看出，猪八戒根本份辨不出悟空扮演的“高三小姐替身”和“高三小姐本人”。客户端分辨不出代理主题对象与真实主题对象，这是代理模式的一个重要用意。

悟空代替高三小姐会见八戒的对象图如下图所示。

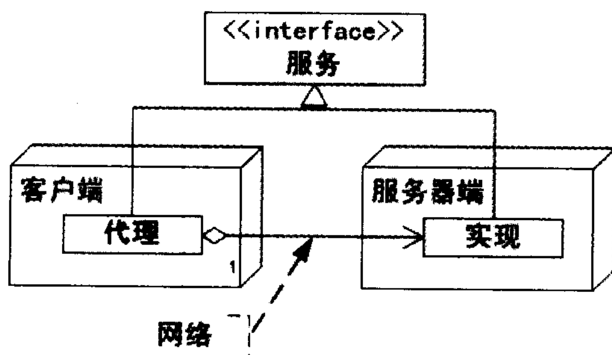


28.7 代理模式的优点和缺点

根据代理种类的不同，代理模式有不同的特点。

远程代理

优点是系统可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。客户完全可以认为被代理的对象是局域的而不是远程的，而代理对象承担了大部分的网络通信工作，远程代理的结构图如下图所示。



由于客户可能没有意识到会启动一个耗费时间的远程调用，因此，客户没有必要的思想准备。

虚拟代理

使用虚拟代理模式的优点就是代理对象可以在必要的时候才将被代理的对象加载。代理可以对加载的过程加以必要的优化。当一个模块的加载十分耗费资源的时候，虚拟代理的优点就非常明显。

在后面的章节中会给出一个详细的例子，读者可以从中学习到虚拟代理的实现方法。

保护代理

保护代理的优点是，它可以在运行时间对用户的有关权限进行检查，然后在核实后决定将调用传递给被代理的对象。

智能引用代理

在访问一个对象时可以执行一些内务处理（Housekeeping）操作，比如计数操作等。在后面的章节中会给出一个例子，供读者学习智能引用代理的实现细节。



28.8　代理模式的实现

代理模式可能并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口。这时候代理对象不能够创建被代理对象，被代理对象必须有系统的其他角色代为创建并且传入。实际上这种做法可以提供更大的灵活性。

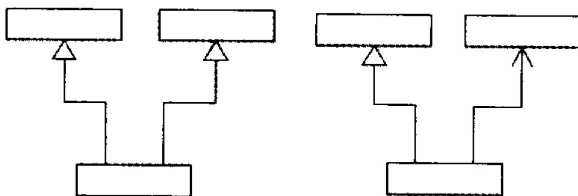
Java 2.0 所提供的对代理模式的支持就是采取了这种做法。

28.9　代理模式与其他模式的关系

代理模式与适配器模式等其他模式有关系。

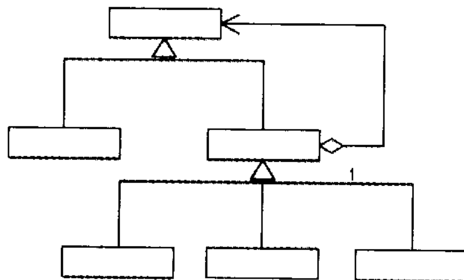
适配器模式

粗看上去，适配器模式与代理模式很相像，它们都可视为一个对象提供一种前置的接口。但是，适配器模式的用意是要改变所考虑的对象的接口，而代理模式并不能改变所代理的对象的接口，在这一点上两个模式有明显的区别。适配器模式的简略类图如下图所示，左边是类的适配器模式，右边是对象的适配器模式。



装饰模式

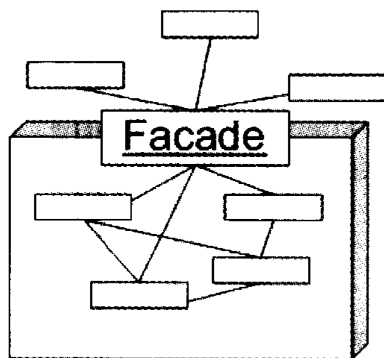
装饰模式与所装饰的对象具有相同的接口，因此这两种模式也有可能混淆。但是，装饰模式应当为所装饰的对象提供增强功能；而代理模式对对象的使用施加控制，并不提供对象本身的增强功能。其简略类图如下图所示。



门面模式

有的时候，门面模式兼任代理的责任。门面对象可能是一个位于另一个地址空间的子系统的远程代理；有的时候，门面模式兼任保护代理的角色，检查调用者的权限；有的时候，门面模式负责记录子系统被调用的次数，因此兼任智能引用代理的角色；有的时候，门面模式兼任虚拟代理的角色，特别是当子系统的加载耗费时间和资源的时候。

这种时候，门面模式又叫做代理门面模式，或门面代理模式，其类图如下图所示。



问答题

1. 有一位同学请求另一位同学代替他参加一个英文水准考试。请问这是什么模式的应用？请给出模拟系统的类图。

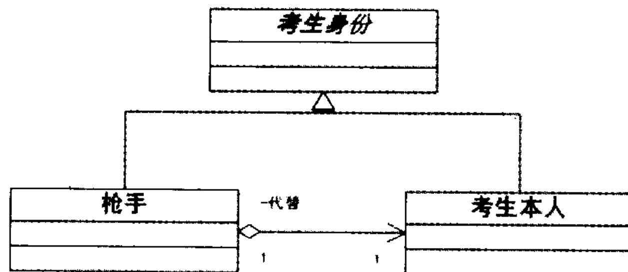
2. 请问“借刀杀人”是什么模式？

3. java.lang.reflect.Proxy 类有一个私有的默认构造子、一个接收参量的保护构造子、以及一个提供本身实例的静态方法 newProxyInstance()，请问这是单例模式还是多例模式？

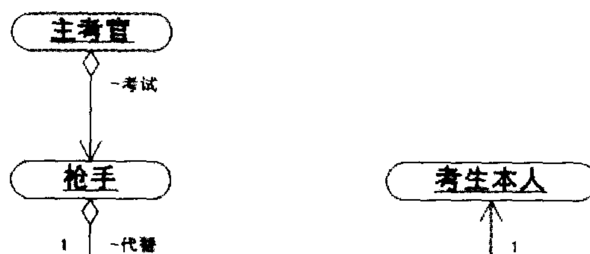
问答题答案

1. 这是代理模式的应用。代考的“枪手”是代理主题角色，而请人代考的同学就是真实主题角色。准考证上的身份就成了抽象主题角色。

模拟系统的类图如下图所示。



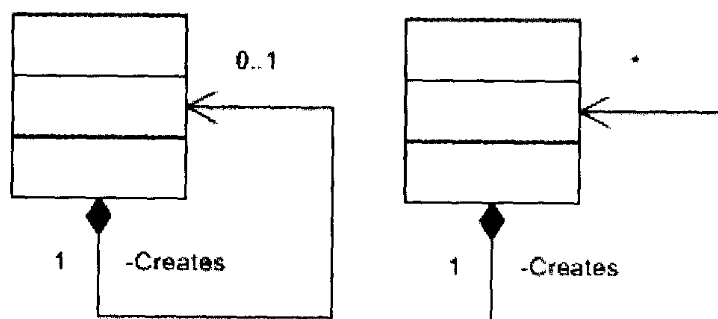
模拟系统的对象图如下图所示。



2. 这是代理模式的应用。借刀杀人的主谋是真实主题，杀手是代理主题。

3. 这不是单例模式的应用。Proxy 类并不能保证只有一个自身的实例。每一次调用 newProxyInstance()方法都会导致一个新的实例被创建出来。

这也不是多例模式。多例模式要求一个类所提供的实例数目有一个固定的上限。显然，Proxy 类可以提供任意多个的实例，理论上没有任何的上限。单例类（左）和多例类（右）的类图如下图所示。



第 29 章　专题：智能引用代理

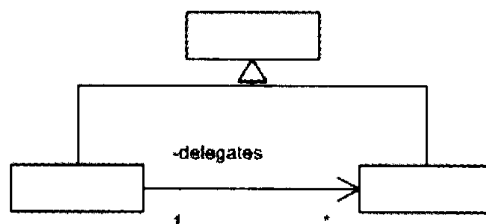
在阅读本章之前，请读者首先阅读本书的“代理（Proxy）模式”一章。这个问题来自一个真实的华尔街金融网站项目。本书对无关的细节做了简化处理，以便让读者将精力集中在设计方案的优化上。

29.1　问　　题

这个动态网站提供美国上市企业的股票买卖信息。一个上市公司往往想知道自己的股票是否被不怀好意者大量收购或者抛售，或者想知道自己的股票被什么基金大量持股。

在本网站，用户做的事情基本上就是搜索和查询。由于网站提供的信息是极有价值的商业信息，因此本网站是一个 B2B 的收费网站。在这里收费意味着两件事：一是必须对用户进行身份检查，二是要对用户的使用次数进行统计，以便按照收费标准进行收费。

正如本章在前面所说，代理模式依照目的可以分为几种，其中就有保护（Protect or Access）代理和智能引用（Smart Reference）代理两种。代理模式的简略类图如下图所示。



一个代理对象对调用被代理对象的用户进行权限检查，并且只允许有适当权限的用户调用原对象，这样的代理对象叫做保护代理。所谓的智能引用代理，就是一个代理对象对调用原对象的用户进行统计计数。

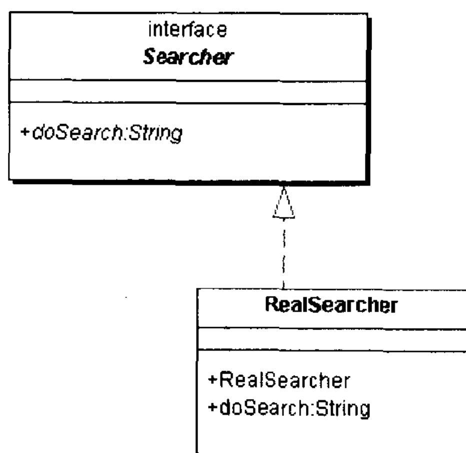
显然，这两种代理模式恰好是本系统所需要的。

29.2　系 统 设 计

第一步：抽象和具体的划分

首先，系统需要一个能进行查询的类。但是有经验的设计师不会仅仅给出一个具体类就交差，因为设计师必须考虑到查询功能在今后可能会有的变化。因此，一个好的设计必须使系统可以在最小的改动下进行最大的扩充。

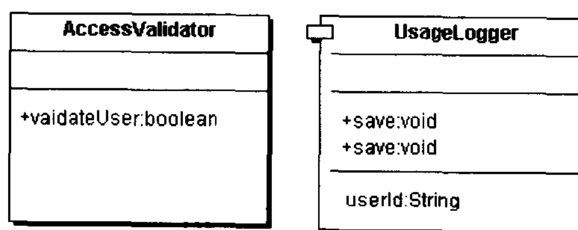
读过本书“开–闭原则（OCP）”一章的读者会想到，设计应当有一个抽象角色和一个具体角色。抽象角色给出接口的声明，而将实现的细节交给具体角色。这样，如果在将来需要有新的查询功能，只需要创建一个新的具体角色即可，而无需改变抽象角色。由于系统知道的角色是抽象角色，因此系统可以在没有代码改变的情况下引入一个新的查询功能。初步的设计图如下图所示。



在这个类图里面，抽象角色由一个 Java 接口（即 Searcher）扮演，而具体角色则由一个具体 Java 类（即 RealSearcher）扮演。

第二步：辅助功能的提供

当然，系统必须有两个辅助对象：一个用做检查用户权限，称为 AccessValidator；另一个用做查询的计数器，称为 UsageLogger，如下图所示。





UsageLogger 类有两个具有不同特征的 save()方法，这是多态性的体现。

这两个辅助对象分别提供适当的方法检查用户的权限和将查询事件记录到数据库中。一般而言，稍有经验的设计师都会想到使用独立的对象负责这两项责任，而不是由负责查询的 Searcher 或者 RealSearcher 负责所有的责任。

第三步：引入代理角色

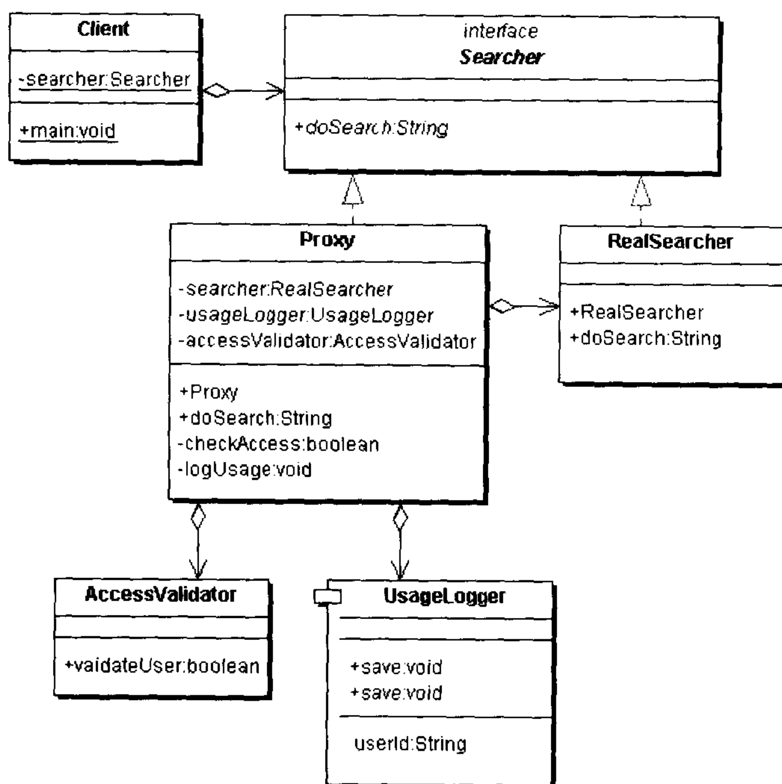
基本的功能现在都由确定的对象提供了：查询功能由 RealSearcher 对象提供；检查用户权限的功能由 AccessValidator 提供；记录查询次数的功能由 UsageLogger 对象提供。

一个没有代理模式知识和应用代理模式经验的设计师往往就在这里停下来了，因为人们往往会认为只要修改一下 RealSearcher 对象的方法，在查询操作之前进行权限的检查，在查询之后进行查询次数的记录就可以了。

当然是可以的，只是不够好。为什么呢？因为 RealSearcher 对象检查对自己的调用不够安全，这个检查应当在 RealSearcher 对象之外的某个地方；而且 RealSearcher 对象可能会有不止一个方法需要进行调用权限的检查，每一个方法都需要自行检查，这会使检查的逻辑出现在数个地方，易于出错而不易维护。这是其一。

其二，对权限的检查和记录调用次数的责任属于对 RealSearcher 对象的调用控制，这个责任应当与 RealSearcher 对象本身分开。换言之，系统应当为 RealSearcher 对象提供一个代理对象，由代理对象负责这两项工作，并将调用委派给 RealSearcher 对象。

因此，就出现了如下图所示的设计图。



代理对象 Proxy 持有 RealSearcher 对象、AccessValidator 对象和 UsageLogger 对象的引用。

针对抽象编程

客户端调用的是 Proxy 对象，如代码清单 1 所示。

代码清单 1：调用代理对象的源代码

```
Searcher searcher = new Proxy();
```

可以看出，变量 searcher 的静态类型是 Searcher，真实类型是 Proxy，这就保证了用户端的调用不能分辨 RealSearcher 和 Proxy。记住，这是代理模式的用意之一，也是“要针对抽象编程，不要针对具体编程”原则的具体体现。

在代理对象被调用时，会在将调用传递给 RealSearcher 之前首先调用私有方法 checkAccess()，以便确认用户具有合适权限；在将调用传递给 RealSearcher 之后调用私有方法 logUsage()，以便将使用次数记录到数据库中。checkAccess()方法会调用 AccessValidator 对象，而 logUsage()会调用 UsageLogger 对象。

29.3　系统的时序

如下图所示，此时序图清楚地给出了调用发生的顺序：

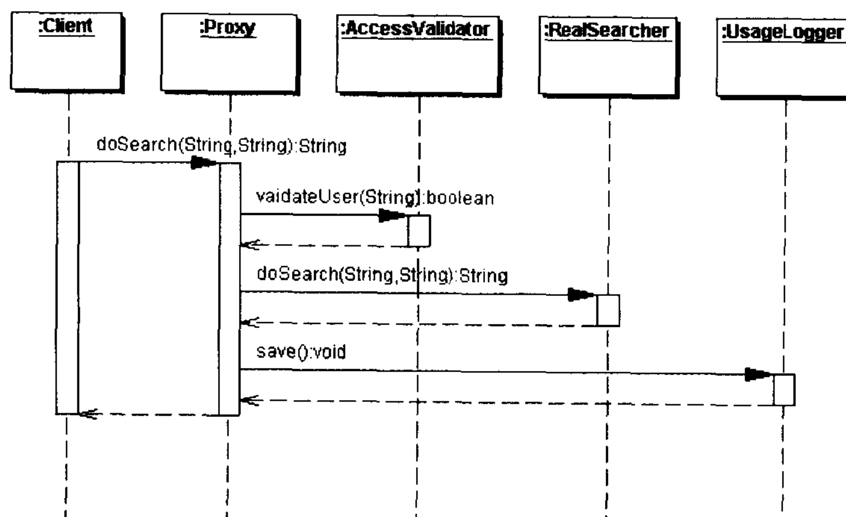
（1）客户端调用代理对象；

（2）代理对象调用 AccessValidator 对象，确定用户确实具有相应的权限；

（3）代理对象调用 RealSearcher 对象，完成查询功能；

（4）代理对象调用 UsageLogger 对象，完成计数功能；

（5）代理对象将 RealSearcher 对象的查询结果返回给客户端。



29.4　系统的源代码

下面给出本系统的源代码，首先是客户端的源代码，如代码清单 2 所示。

代码清单 2：客户端的源代码

```
package com.javapatterns.proxy.smartproxy;
public class Client
{
    // 声明一个静态类型为 Searcher 的静态变量
    private static Searcher searcher;
    public static void main(String[] args)
    {
        // 此静态变量的真实类型为 Proxy
```

```
            searcher = new Proxy();
            String userId = "Admin";
            String searchType = "SEARCH_BY_ACCOUNT_NUMBER";
            String result =
                searcher.doSearch(userId, searchType);
            System.out.println(result);
        }
    }
```

下面是抽象主题角色的源代码，如代码清单 3 所示。抽象主题角色规定了所有的具体主题和代理主题都必须实现的接口，即 doSearch()方法。

代码清单 3：抽象主题角色的源代码

```
package com.javapatterns.proxy.smartproxy;
public interface Searcher
{
    /**
     *   声明一个抽象方法
     */
    String doSearch(String userId, String searchType);
}
```

代理角色的源代码如代码清单 4 所示。可以看出，代理角色不是单纯地将调用传递给具体主题角色，而是进行了委派前的权限查询和委派后的次数记录。

代码清单 4：代理角色的源代码

```
package com.javapatterns.proxy.smartproxy;
public class Proxy implements Searcher
{

    private RealSearcher searcher;
    private UsageLogger usageLogger;
    private AccessValidator accessValidator;
    /**
     *   构造子
     */
    public Proxy()
    {
        searcher = new RealSearcher();
    }
    /**
     *   实现查询操作
     */
    public String doSearch(String userId,
        String keyValue)
    {
        if (checkAccess(userId))
```

```
        {
            String result =
                searcher.doSearch(null, keyValue);
            logUsage(userId);
            return result;
        }
        else
        {
            return null;
        }
    }
    /**
     *   查询前的权限操作
     */
    private boolean checkAccess(String userId)
    {
        accessValidator = new AccessValidator();
        return accessValidator.vaidateUser(userId);
    }
    /**
     *   查询后的日志操作
     */
    private void logUsage(String userId)
    {
        UsageLogger logger = new UsageLogger();
        logger.setUserId(userId);
        logger.save();
    }
}
```

下面给出 RealSearcher 类的示意性源代码，如代码清单 5 所示。在一个真实系统中，doSearch()方法应当对数据库进行查询，并返回结果。

代码清单 5：真实主题的源代码

```
package com.javapatterns.proxy.smartproxy;
class RealSearcher implements Searcher
{
    /**
     *   构造子
     */
    public RealSearcher()
    {
    }
    /**
     *   真实的查询工作在这里发生
     */
    public String doSearch(String userId, String keyValue)
```

```
    {
        String sql = "SELECT * FROM data_table WHERE key_col = '"
            + keyValue + "'";
        // execute this SQL Statement and
        // concatenate a result string
        return "result set";
    }
}
```

下面给出检查权限用的 UsageLogger 类的示意性源代码，如代码清单 6 所示。在一个真实系统中，validateUser()方法应当调用数据库方法，并且在从数据库中得到数据的基础之上做出判断。

代码清单 6：检查权限的对象 AccessValidator 的源代码

```
package com.javapatterns.proxy.smartproxy;
public class AccessValidator
{
    /**
     *   用户权限检查发生在这里
     */
    public boolean vaidateUser(String userId)
    {
        if (userId.equals("Admin"))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

下面给出记录次数用的 UsageLogger 类的示意性源代码，如代码清单 7 所示。在一个真实系统中，save()方法应当调用数据库方法，执行这里给出的 SQL 语句。

代码清单 7：记录次数的 UsageLogger 类的源代码

```
package com.javapatterns.proxy.smartproxy;
public class UsageLogger
{
    private String userId;
    /**
     *   用户 ID 的赋值方法
     */
    public void setUserId(String userId)
    {
        this.userId = userId;
```

```
    }
    /**
     *    将这次使用记录加到日志中
     */
    public void save()
    {
        String sql = "INSERT INTO USAGE_TABLE (user_id) "
            + " VALUES(" + userId + ")";
        //execute this SQL statement
    }
    /**
     *    将这次使用记录加到日志中
     */
    public void save(String userId)
    {
        this.userId = userId;
        save();
    }
}
```

本章演示了怎样利用代理模式设计一个检查用户权限和记录用户使用次数的子系统。

在实际系统中，用户权限检查要设计到用户是否已经付过使用费用，并且考虑到有些用户是免费邀请来观摩试用的礼宾用户。

虽然使用次数记录较为简单，但是当统计用户使用次数并生成费用账单时，逻辑也会很复杂。典型的逻辑如：一个用户如果在一个小时内进行完全一样的查询的话，只按一次计费。

第 30 章　专题：虚拟代理的例子

请读者在阅读本章之前，首先阅读本书的“代理（Proxy）模式”一章。

30.1　问　　题

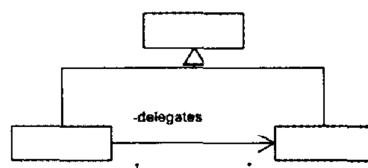
加载延缓

相信绝大多数的读者都见过这样的情况，一个系统需要加载一个模块，但是此模块的加载要耗费相当长的时间，因此，系统要显示一段“正在加载”的信息，同时将模块加载。在模块加载后，系统将“正在加载”的文字取消掉，并启动此模块。常见的例子包括 Netscape 浏览器、JBuilder 环境等中大型软件。

有的时候，加载一个图像也会造成时间的延迟，因此，系统需要在放置图像的地方放一段文字信息，请用户耐心等待。系统会在另一个线程中加载图像，而一旦图像加载完毕，就将等待信息换成该图像。这种做法叫做加载延缓，加载延缓显然是比较友好的用户界面设计方案。

虚拟代理模式的应用

当一个真实主题对象的加载需要耗费资源时，一个虚拟代理对象可以代替真实对象接受请求。一旦接到请求，代理对象马上打出一段“正在加载”的信息，并在适当的时候加载真实主题对象，也就是模块或者图像。代理模式的简略类图如右图所示。



30.2　系统的要求

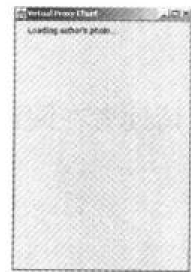
系统的要求

本章将给出一个加载图像的例子，由于图像的加载会耗费一定的资源，因此，要求设计一个虚拟代理对象，以替代图像对象接受客户端的请求。当虚拟代理对象接到请求后，会按照预定的逻辑首先显示一段等待信息，然后在另一个线程中加载图像。当图像加载完

成后，主线程会决定将图像显示出来。

这个例子是建立在一个例子的基础之上的[GEARY02]，并有所改动（不仅仅改变了张灵长类的照片）。本书鼓励读者阅读文献[GEARY02]原文。

系统运行情况



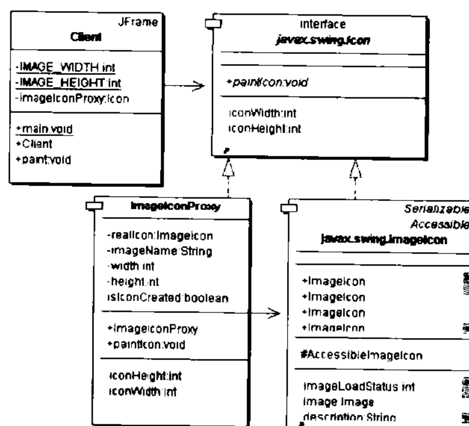
系统的运行情况如下：系统启动时，会首先显示一段等待信息，与此同时，会决定何时加载和显示图像，如右图所示。

当程序认为合适的时候，就会将图像加载并显示给用户。如下图所示。



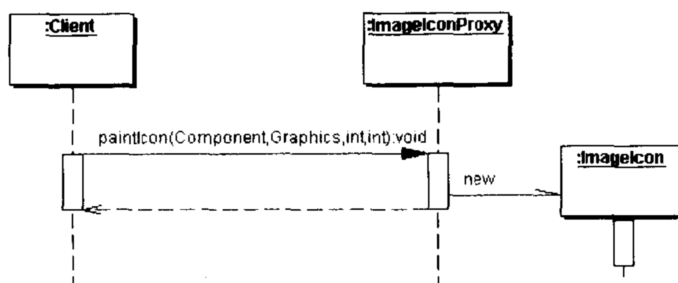
30.3 系统的设计

系统由一个 JFrame 对象、一个 Icon 对象以及此 Icon 对象的虚拟代理对象组成。下图所示的类图显示了三者之间的关系。

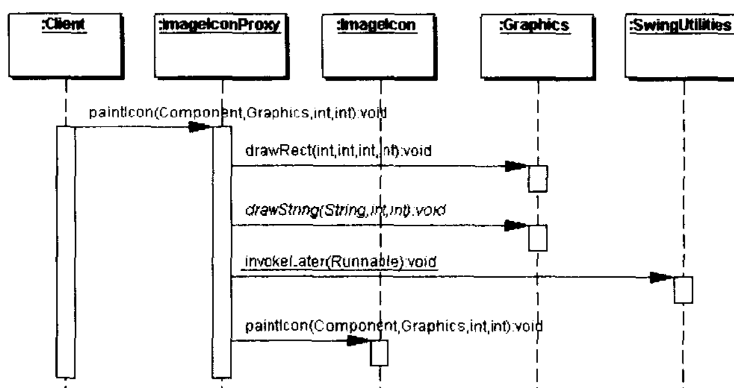


30.4 系统的时序

系统的活动序列图更能够反映此系统的动态关系。下面就是在本系统的活动过程中各个角色的相互作用发生的时间序列图，如下图所示，这里仅给出了三个主要的角色。



可以看出，系统的客户端对象调用代理 ImageIconProxy 对象，而此代理则负责将调用传递给真实主题角色，即一个 ImageIcon 对象，如下图所示。



系统的客户端对象调用代理 ImageIconProxy 对象时，此代理对象调用所传入的 Graphics 对象的 drawRec()方法，显示一段"Loading author's photo..."的等待信息。然后，代理对象创建一个内部线程对象，并将之传入 SwingUtilities 的 invokeLater()方法中，让它负责加载真实主题。至此，代理对象成功地延迟了真实主题的加载。随后，代理主题认为显示真实主题的时机成熟了，于是将真实主题（也就是图像）显示出来。

30.5　系统的源代码

首先给出客户端的源代码，如代码清单 1 所示。

代码清单 1：客户端的源代码

```
package com.javapatterns.proxy.imageloader;
import java.awt.Graphics;
import java.awt.Insets;
import javax.swing.JFrame;
import javax.swing.Icon;
public class Client extends JFrame
{
    private static int IMAGE_WIDTH = 270;
    private static int IMAGE_HEIGHT = 380;
    private Icon imageIconProxy = null;
    static public void main(String[] args)
```

```
    {
        Client app = new Client();
        app.show();
    }
    /**
     *   构造子
     */
    public Client()
    {
        super("Virtual Proxy Client");
        imageIconProxy = new ImageIconProxy(
            "c:/hongyan.jpg",
            IMAGE_WIDTH,
            IMAGE_HEIGHT);
        setBounds(100, 100, IMAGE_WIDTH + 10,
                IMAGE_HEIGHT + 30);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    /**
     *   置换掉 java.awt.Container 的方法
     */
    public void paint(Graphics g)
    {
        super.paint(g);
        Insets insets = getInsets();
        imageIconProxy.paintIcon(this, g,
            insets.left , insets.top);
    }
}
```

在运行这个程序时，需要有 c:/hongyan.jpg 图像文件。

在运行客户端类的时候，注意在 C 盘上要有一张名为 hongyan.jpg 的图像文件存在。

下面给出代理对象 ImageIconProxy 的源代码，如代码清单 2 所示。它的 paintIcon()方法被重载，以便在图像加载过程中打印出“Loading author's photo...”的等待信息。同时此方法会在合适的时候加载图像，并在图像的下方打出“Java and Patterns by Jeff Yan, Ph.D”的字样。

代码清单 2：代理角色 ImageIconProxy 的源代码

```
package com.javapatterns.proxy.imageloader;
import java.awt.Graphics;
import java.awt.Component;
import javax.swing.ImageIcon;
import javax.swing.Icon;
import javax.swing.SwingUtilities;
public class ImageIconProxy implements Icon
{
```

```
private ImageIcon realIcon = null;
private String imageName;
private int width;
private int height;
boolean isIconCreated = false;
 /**
  *   构造子
  */
 public ImageIconProxy(String imageName,
      int width, int height)
 {
    this.imageName = imageName;
    this.width = width;
    this.height = height;
 }
 /**
  *   图像高度的取值方法
  */
 public int getIconHeight()
 {
      return realIcon.getIconHeight();
 }
 /**
  *   图像宽度的取值方法
  */
 public int getIconWidth()
 {
        return realIcon.getIconWidth();
 }
 /**
  *   加载图像
  */
 public void paintIcon(final Component c,
      Graphics g, int x, int y)
 {
    if(isIconCreated)
    {
       realIcon.paintIcon(c, g, x, y);
       g.drawString("Java and Patterns by Jeff Yan, Ph.D",
            x+20, y+370);
    }
    else
    {
       g.drawRect(x, y, width-1, height-1);
       g.drawString("Loading author's photo...",
            x+20, y+20);
```

```
                // 图像在另外一个线程中被加载
                synchronized(this)
                {
                    SwingUtilities.invokeLater(new Runnable()
                    {
                        public void run()
                        {
                            try
                            {
                                // 减缓图像的加载过程
                                Thread.currentThread().sleep(2000);
                                // 将图像加载
                                realIcon = new ImageIcon(imageName);
                                isIconCreated = true;
                            }
                            catch(InterruptedException ex)
                            {
                                ex.printStackTrace();
                            }
                            // 当图像被加载后，重新描绘视窗构件
                            c.repaint();
                        }
                    });
                }
            }
        }
    }
```

在上面使用 SwingUtilities.invokeLater()方法开启一个独立的新线程，用以执行一个新的任务。这个新的任务就是首先休眠 2 秒钟，然后加载图像。在加载完成后，重新描绘视窗构件。

SwingUtilities 另有一个与 invokeLater()方法相似的方法——invokeAndWait()。后者与前者不同的是，它会将主线程的执行封锁住，直到新线程的任务完成为止。

本章演示了如何利用代理模式实现图像加载时会遇到的加载延缓处理。在实际系统中，加载延缓往往是针对较大型的软件模块，而不仅仅是一张图像。但是实际做法可以是一样的：利用虚拟代理对象接受加载请求，并将真实主题的加载延缓到真正需要的时候。

参考文献

[GEARY02] David Geary. Take Control with the Proxy Design Pattern. JavaWorld, http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html，February，2002

[OAKS99] Scott Oaks and Henry Wong. Java Threads (2nd Edition). published by O'Reilly，January 1999

第 31 章　享元模式（Flyweight Pattern）

本章讨论 Flyweight 模式。Flyweight 在拳击比赛中指最轻量级，即“蝇量级”，有些作者翻译为“羽量级”。本书选择使用“享元模式”的意译，是因为这样更能反映模式的用意，这一译名最早是由文献[GOF95Z]提出的。

31.1　引　　言

享元模式的用意

享元模式是对象的结构模式[GOF95]。享元模式以共享的方式高效地支持大量的细粒度对象。

享元对象能做到共享的关键是区分内蕴状态（Internal State）和外蕴状态（External State）。

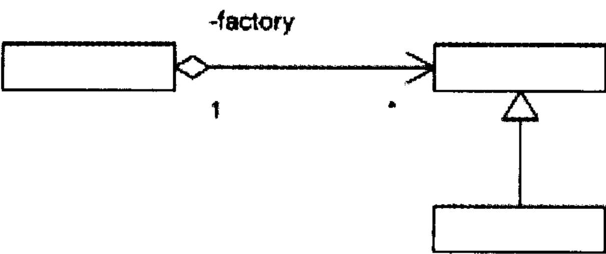
一个内蕴状态是存储在享元对象内部的，并且是不会随环境改变而有所不同的。因此，一个享元可以具有内蕴状态并可以共享。

一个外蕴状态是随环境改变而改变的、不可以共享的状态。享元对象的外蕴状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入到享元对象内部。

外蕴状态不可以影响享元对象的内蕴状态。换句话说，它们是相互独立的。

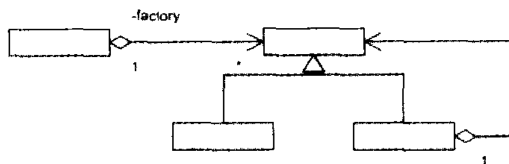
享元模式的种类

根据所涉及的享元对象的内部表象，享元模式可以分成单纯享元模式和复合享元模式两种形式。下图所示是单纯享元模式的结构示意图。



下图所示是复合享元模式的结构示意图。





从上面的类图可以看出，在复合享元模式中，享元对象构成合成模式。因此，复合享元模式实际上是单纯享元模式与合成模式的组合。

享元模式的应用

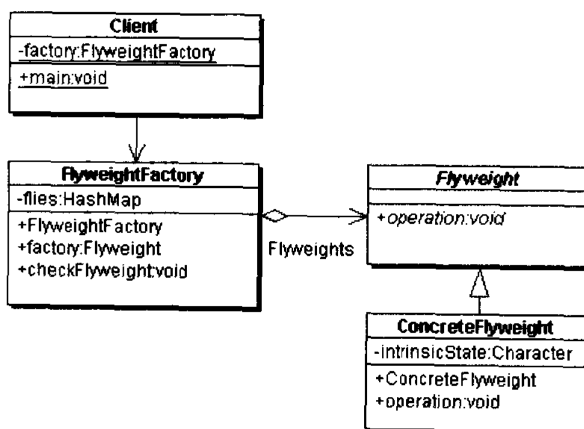
享元模式在编辑器系统中大量使用。一个文本编辑器往往会提供很多种字体，而通常的做法就是将每一个字母做成一个享元对象。享元对象的内蕴状态就是这个字母，而字母在文本中的位置和字模风格等其他信息则是外蕴状态。比如，字母 a 可能出现在文本的很多地方，虽然这些字母 a 的位置和字模风格不同，但是所有这些地方使用的都是同一个字母对象。这样一来，字母对象就可以在整个系统中共享。

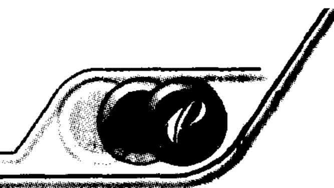
在 Java 语言中，String 类型就使用了享元模式。String 对象是不变对象，一旦创建出来就不能改变。如果需要改变一个字符串的值，就只好创建一个新的 String 对象。在 JVM 内部，String 对象都是共享的。如果一个系统中有两个 String 对象所包含的字符串相同的话，JVM 实际上只创建一个 String 对象提供给两个引用，从而实现 String 对象的共享。String 的 intern()方法给出这个字符串在共享池中的惟一实例。

31.2 单纯享元模式的结构

类图与角色

在单纯享元模式中，所有的享元对象都是可以共享的。下面给出一个单纯享元模式的简单实现，以说明单纯享元模式的结构，如下图所示。





单纯享元模式所涉及的角色如下：

- 抽象享元角色：此角色是所有的具体享元类的超类，为这些类规定出需要实现的公共接口。那些需要外蕴状态（External State）的操作可以通过调用商业方法以参数形式传入。
- 具体享元（ConcreteFlyweight）角色：实现抽象享元角色所规定的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。享元对象的内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享的。
- 享元工厂（FlyweightFactory）角色：本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有一个复合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。
- 客户端（Client）角色：本角色需要维护一个对所有享元对象的引用。本角色需要自行存储所有享元对象的外蕴状态。

示意性源代码

抽象享元角色的源代码如代码清单 1 所示。可以看出，抽象享元角色声明了operation(state)方法，其中 state 参量代表享元对象的外蕴状态。

代码清单 1：抽象享元角色 Flyweight 类的源代码

```
package com.javapatterns.flyweight.simple;
abstract public class Flyweight
{
    // 一个示意性的方法，参数 state 是外蕴状态
    abstract public void operation(String state);
}
```

下面是具体享元类的源代码，如代码清单 2 所示。具体享元类实现了抽象享元角色所给定的接口，在这里就是 operation()方法，享元角色可以具有内蕴状态。在这里享元对象有一个内蕴状态，以一个 Character 类型的 intrinsicState 属性代表，它的值应当在享元对象被创建时赋予。所有的内蕴状态在对象创建之后，就不会再改变了。

虽然这里的内蕴状态是 Character 类型的，但是模式对内蕴状态的类型并无限制。

如果一个享元对象有外蕴状态的话，所有的外部状态都必须存储在客户端，在使用享元对象时，再由客户端传入享元对象。这里只有一个外蕴状态，operation()方法的参量 state 就是由外部传入的外蕴状态。

代码清单 2：具体享元角色 ConcreteFlyweight 类的源代码

```
package com.javapatterns.flyweight.simple;
public class ConcreteFlyweight extends Flyweight
{
    private Character intrinsicState = null;
```

```
    /**
     *  构造子，内蕴状态作为参量传入
     */
    public ConcreteFlyweight(Character state)
    {
        this.intrinsicState = state;
    }

    /**
     *  外蕴状态作为参量传入方法中，改变方法的行为，
     *  但是并不改变对象的内蕴状态
     */
    public void operation(String state)
    {
        System.out.print( "\nIntrinsic State = "
            + intrinsicState +
            ", Extrinsic State = " + state);
    }
}
```

必须指出的是，客户端不可以直接将具体享元类实例化，而必须通过一个工厂对象，利用一个 factory()方法得到享元对象。一般而言，享元工厂对象在整个系统中只有一个，因此可以使用单例模式。

当客户端需要单纯享元对象的时候，需要调用享元工厂的 factory()方法，并传入所需的单纯享元对象的内蕴状态，由工厂方法产生所需的享元对象。请参见下面给出的示意性客户端角色的源代码。

享元工厂 FlyweightFactory 的源代码，如代码清单 3 所示。

代码清单 3：享元工厂角色 FlyweightFactory 类的源代码

```
package com.javapatterns.flyweight.simple;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class FlyweightFactory
{
    private HashMap flies = new HashMap();
    private Flyweight lnkFlyweight;
   /**
    *  默认构造子
    */
    public FlyweightFactory(){ }

   /**
    *  构造子，内蕴状态作为参量传入
    */
    public Flyweight factory(Character state)
```

```
    {
        if ( flies.containsKey( state ) )
        {
            return (Flyweight) flies.get( state );
        }
        else
        {
            Flyweight fly = new ConcreteFlyweight(state);
            flies.put( state , fly);
            return fly;
        }
    }

    /**
     *  辅助方法
     */
    public void checkFlyweight()
    {
        Flyweight fly ;
        int i = 0;
        System.out.println("\n======checkFlyweight()========");
        for ( Iterator it = flies.entrySet().iterator() ;
            it.hasNext() ;  )
        {
            Map.Entry e = (Map.Entry) it.next();
            System.out.println("Item " + (++i)
                + " : " + e.getKey());
        }
        System.out.println("======checkFlyweight()========");
    }
}
```

在使用这种享元模式的时候，首先需要创建享元工厂对象，然后向享元工厂对象要求具有某个状态的享元，如代码清单 4 所示。

代码清单 4：抽象享元角色 Flyweight 类的源代码

```
// 创建一个享元工厂对象
FlyweightFactory factory = new FlyweightFactory();
// 向享元工厂对象请求一个内蕴状态为'a'的享元对象
Flyweight fly = factory.factory(new Character('a'));
// 以参量方式传入一个外蕴状态
fly.operation("First Call");
// 向享元工厂对象请求一个内蕴状态为'b'的享元对象
fly = factory.factory(new Character('b'));
// 以参量方式传入一个外蕴状态
fly.operation("Second Call");
// 向享元工厂对象请求一个内蕴状态为'a'的享元对象
```

```
fly = factory.factory(new Character('a'));
// 以参量方式传入一个外蕴状态
fly.operation("Third Call");
```

虽然上面申请了三个享元对象，但是实际上创建的享元对象只有两个，这就是共享的含义。

享元工厂角色的 checkFlyweight()可方法按照代码清单 5 所示的方法使用。

代码清单 5：调用享元工厂角色的 checkFlyweight()方法

```
factory.checkFlyweight();
```

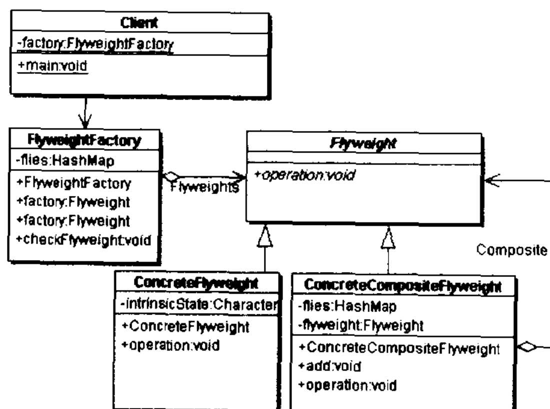
当 checkFlyweight()方法被调用时，会打印出所有的独立的享元对象，为系统设计过程提供辅助信息。在上面的例子里，如果调用这个方法，会给出两个享元角色，分别对应于“a”和“b”两种内蕴状态。

31.3 复合享元模式的结构

类图和角色

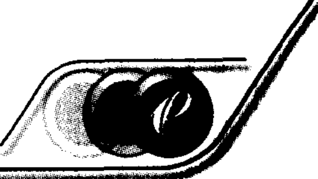
在上面的单纯享元模式中，所有的享元对象都是单纯享元对象，也就是说都是可以直接共享的。下面考虑一个较为复杂的情况，即将一些单纯享元使用合成模式加以复合，形成复合享元对象。这样的复合享元对象本身不能共享，但是它们可以分解成单纯享元对象，而后者则可以共享。

复合享元模式的类图如下图所示。



享元模式所涉及的角色有抽象享元角色、具体享元角色、复合享元角色、享员工厂角色，以及客户端角色等。

- 抽象享元角色：此角色是所有的具体享元类的超类，为这些类规定出需要实现的公共接口。那些需要外蕴状态（External State）的操作可以通过方法的参数传入。抽象享元的接口使得享元变得可能，但是并不强制子类实行共享，因此并非所有



的享元对象都是可以共享的。

- 具体享元（ConcreteFlyweight）角色：实现抽象享元角色所规定的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。享元对象的内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享。有时候具体享元角色又叫做单纯具体享元角色，因为复合享元角色是由单纯具体享元角色通过复合而成的。
- 复合享元（UnsharableFlyweight）角色：复合享元角色所代表的对象是不可以共享的，但是一个复合享元对象可以分解成为多个本身是单纯享元对象的组合。复合享元角色又称做不可共享的享元对象。
- 享元工厂（FlyweightFactory）角色：本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象请求一个享元对象的时候，享元工厂角色需要检查系统中是否已经有一个符合要求的享元对象，如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个新的合适的享元对象。
- 客户端（Client）角色：本角色还需要自行存储所有享元对象的外蕴状态。

示意性源代码

下面这些角色示意性实现的源代码。首先是抽象享元角色的源代码，如代码清单 6 所示。

代码清单 6：抽象享元角色 Flyweight 类的源代码

```
package com.javapatterns.flyweight.composite;
abstract public class Flyweight
{
    /**
     * 外蕴状态作为参量传入到方法中
     */
    abstract public void operation(String state);
}
```

具体享元对象的源代码，如代码清单 7 所示，具体享元角色的主要责任有两个：

- 实现了抽象享元角色所声明的接口，也就是 operation()方法。operation()方法接收一个外蕴状态作为参量。
- 为内蕴状态提供存储空间，在本实现中就是 intrinsicState 属性。享元模式本身对内蕴状态的存储类型并无要求，这里的内蕴状态是 Character 类型，是为了给复合享元的内蕴状态选做 String 类型提供方便。

代码清单 7：具体享元角色 ConcreteFlyweight 类的源代码

```
package com.javapatterns.flyweight.composite;
public class ConcreteFlyweight extends Flyweight
{
```

```
    private Character intrinsicState = null;
   /**
    *  构造子，内蕴状态作为参量传入
    */
    public ConcreteFlyweight(Character state)
    {
        this.intrinsicState = state;
    }
   /**
    *  外蕴状态作为参量传入到方法中
    */
    public void operation(String state)
    {
        System.out.print( "\nInternal State = " +
            intrinsicState + " Extrinsic State = "
            + state );
    }
}
```

具体复合享元角色的源代码，如代码清单 8 所示，它有以下两个责任：

- 复合享元对象是由单纯的享元对象通过复合而成，因此它提供了 add()这样的聚集管理方法。由于一个复合享元对象具有不同的聚集元素，这些聚集元素在复合享元对象被创建之后加入，这本身就意味着复合享元对象的状态是会改变的，因此复合享元对象是不能共享的。
- 复合享元角色实现了抽象享元角色所规定的接口，也就是 operation()方法。这个方法有一个参量，代表复合享元对象的外蕴状态。一个复合享元对象的所有单纯享元对象元素的外蕴状态都是与复合享元对象的外蕴状态相等的；而一个复合享元对象所含有的单纯享元对象的内蕴状态一般是不相等的，不然就没有使用价值了。

代码清单 8：具体复合享元角色 UnsharedConcreteFlyweight 类的源代码

```
package com.javapatterns.flyweight.composite;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class ConcreteCompositeFlyweight extends Flyweight
{
    private HashMap flies = new HashMap(10);
    private Flyweight flyweight;
   /**
    *  默认构造子
    */
    public ConcreteCompositeFlyweight()
    {
    }
```

```
    /**
     *  增加一个新的单纯享元对象到聚集中
     */
    public void add(Character key, Flyweight fly)
    {
        flies.put(key, fly);
    }

    /**
     *  外蕴状态作为参量传入到方法中
     */
    public void operation(String extrinsicState)
    {
        Flyweight fly =   null;
        for ( Iterator it = flies.entrySet().iterator() ;
            it.hasNext() ;  )
        {
            Map.Entry e = (Map.Entry) it.next();
            fly = (Flyweight) e.getValue();
            fly.operation(extrinsicState);
        }
    }
}
```

下面给出享元工厂 FlyweightFactory 的源代码，如代码清单 9 所示。享元工厂在多态性的基础之上提供两种不同的方法，一个用于提供单纯享元对象，另一个用于提供复合享元对象。

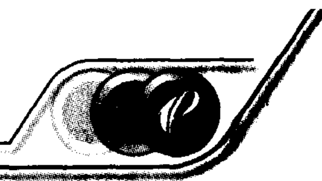
当客户端需要单纯享元对象的时候，需要调用享元工厂的 factory()方法，并传入所需的单纯享元对象的内蕴状态。在这里，内蕴状态是 Character 类型的。

而当客户端需要复合享元对象的时候，需要调用享元工厂的 factory()方法，并传入所需的复合享元对象的所有复合元素（也就是单纯享元对象）的内蕴状态。由于单纯享元对象的内蕴状态是 Character 类型的，因此就可以传入一个 String 类型的参量，作为聚集中所有的元素的内蕴状态的总和。这仅仅是采用 Character 类型作为单纯享元的内蕴状态存储类型的特殊之处。如果读者面对的系统没有这种巧合，则完全可以使用一个聚集（如 Vector）对象作为复合享元的状态传入。

代码清单 9：享元工厂角色 FlyweightFactory 类的源代码

```
package com.javapatterns.flyweight.composite;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class FlyweightFactory
{
    private HashMap flies = new HashMap();
    /**
```

```
 *  默认构造子
 */
 public FlyweightFactory(){ }
/**
 *  复合享元工厂方法，所需状态以参量形式传入
 *  这个参量恰好可以使用 String
 *  类型。读者完全可以使用一个聚集，如 Vector 对象等
 */
 public Flyweight factory(String compositeState)
 {
     ConcreteCompositeFlyweight compositeFly =
         new ConcreteCompositeFlyweight();
     int length = compositeState.length();
     Character state = null;
     for(int i = 0; i < length; i++)
     {
         state = new Character(compositeState.charAt(i));
         System.out.println("factory(" + state + ")");
         compositeFly.add( state, this.factory( state));
     }
     return compositeFly;
 }
/**
 *  单纯享元工厂方法，所需状态以参量形式传入
 */
 public Flyweight factory(Character state)
 {
     // 检查具有此状态的享元是否已经存在
     if ( flies.containsKey( state ) )
     {
         // 具有此状态的享元已经存在，因此直接将它返还
         return (Flyweight) flies.get( state );
     }
     else
     {
         // 具有此状态的享元不存在，因此创建新实例
         Flyweight fly = new ConcreteFlyweight(state);
         // 将实例存储到聚集中
         flies.put( state , fly);
         // 将实例返还
         return fly;
     }
 }
 public void checkFlyweight()
 {
     Flyweight fly ;
```



```
            int i = 0 ;
            System.out.println("\n======checkFlyweight()========");
            for ( Iterator it = flies.entrySet().iterator() ;
                it.hasNext() ;   )
            {
                Map.Entry e = (Map.Entry) it.next();
                System.out.println( "Item " + (++i) + " : "
                    + e.getKey());
            }
            System.out.println("\n======checkFlyweight()========");
        }
    }
```

在使用这种享元模式的时候，首先需要创建享元工厂对象，然后向享元工厂对象要求具有某个内蕴状态的单纯享元，或者具有某个组合状态的复合享元。下面就是创建复合享元对象的方法，如代码清单 10 所示。

代码清单 10：抽象享元角色 Flyweight 类的源代码

```
    Flyweight fly = factory.factory("aba");
    fly.operation("Composite Call");
```

可以看出，上面的程序向享元工厂申请了一个内蕴状态为“aba”的复合享元对象，而且外蕴状态都是“Composite Call”。

checkFlyweight()办法的使用如代码清单 11 所示。

代码清单 11：抽象享元角色 Flyweight 类的源代码

```
    factory.checkFlyweight();
```

如果调用 checkFlyweight()方法的话，会看到系统一共创建了两个单纯享元对象，分别对应于“a”和“b”两种内蕴状态。

31.4　模式的实现

使用不变模式实现享元角色

享元模式里的享元对象不一定非得是不变对象（Immutable），但是很多的享元对象确实被设计成了不变对象。由于不变对象的状态在被创建之后就不再变化，因此不变对象满足享元模式对享元对象的要求。

使用备忘录模式实现享元工厂角色

享元工厂负责维护一个表，通过这个表把很多全同的实例与代表它们的一个对象联系

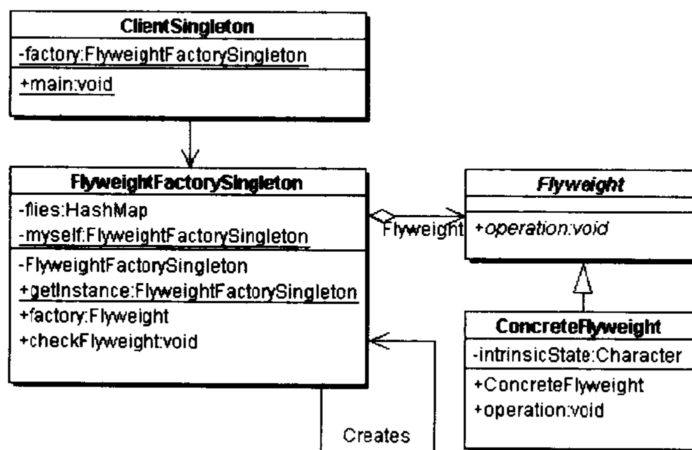
起来。这就是备忘录模式的应用。

使用单例模式实现享元工厂角色

系统往往只需要一个享元工厂的实例，所以享元工厂可以设计成为单例模式。关于单例模式，请阅读本书的“单例模式”一章。

单纯享元模式中的享元工厂角色

使用单例模式改写后的系统设计图如下图所示。



下面就是经过改造的使用单例模式的享元工厂角色的源代码，如代码清单 12 所示。

代码清单 12：在单纯的共享模式中使用单例模式实现共享工厂角色

```
package com.javapatterns.flyweight.simple;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class FlyweightFactorySingleton
{
    private HashMap flies = new HashMap();
    private static FlyweightFactorySingleton myself =
        new FlyweightFactorySingleton();
  /**
   *  构造子是私有的，外界无法直接实例化
   */
   private FlyweightFactorySingleton(){}
  /**
   *  静态工厂方法，向外界提供惟一的实例
   */
   public static FlyweightFactorySingleton getInstance()
   {
       return myself;
   }
  /**
```

```
 *  工厂方法，向外界提供含有指定内蕴状态的对象
 */
public synchronized Flyweight factory(Character state)
{
    // 检查具有此状态的享元是否已经存在
    if ( flies.containsKey( state ) )
    {
        // 具有此状态的享元对象已经存在，因此直接返还此对象
        return (Flyweight) flies.get( state );
    }
    else
    {
        // 具有此状态的享元对象不存在，因此创建新对象
        Flyweight fly = new ConcreteFlyweight( state );
        // 将新对象存储到聚集里
        flies.put( state , fly);
        // 将对象返还
        return fly;
    }
}
/**
 *  辅助方法，打印所有已经创建的享元对象清单
 */
public void checkFlyweight()
{
    Flyweight fly ;
    int i = 0;
    System.out.println("\n======checkFlyweight()=========");
    for ( Iterator it = flies.entrySet().iterator() ;
        it.hasNext() ;  )
    {
        Map.Entry e = (Map.Entry) it.next();
        System.out.println("Item " + (++i) + " : "
            + e.getKey());
    }
    System.out.println("======checkFlyweight()========");
}
}
```

熟悉单例模式的读者可以看出，上面所使用的实现方式是“饿汉”式的单例模式。

下面给出一个示意性客户端类的源代码，如代码清单 13 所示，显示出调用单例工厂对象的 getInstance()方法，以得到具体享元类的实例。

代码清单 13：单纯享元模式中的示意性客户端的源代码

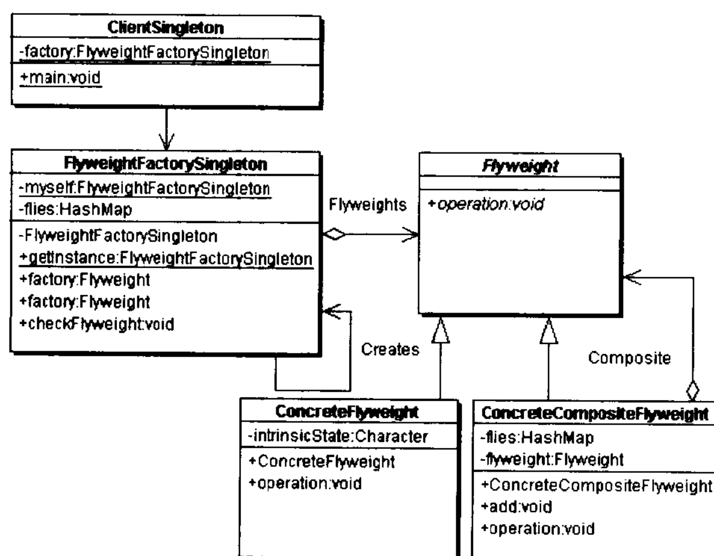
```
package com.javapatterns.flyweight.simple;
public class ClientSingleton
```

```
{
    private static FlyweightFactorySingleton factory;
    static public void main(String[] args)
    {
        // 创建享元工厂对象
        factory = FlyweightFactorySingleton.getInstance();
        // 向享元工厂对象请求一个内蕴状态为'a'的享元对象
        Flyweight fly = factory.factory(new Character('a'));
        // 以参量方式传入外蕴状态
        fly.operation("First Call");
        // 向享元工厂对象请求一个内蕴状态为'b'的享元对象
        fly = factory.factory(new Character('b'));
        // 以参量方式传入外蕴状态
        fly.operation("Second Call");
        // 向享元工厂对象请求一个内蕴状态为'a'的享元对象
        fly = factory.factory(new Character('a'));
        // 以参量方式传入外蕴状态
        fly.operation("Third Call");
        // 调用享元工厂对象的辅助方法，
        // 查看到底有几个享元对象被创建
        factory.checkFlyweight();
    }
}
```

如果调用 checkFlyweight()方法，会看到系统一共创建了两个享元对象，分别对应于“a”和“b”两种内蕴状态。

复合享元模式中的享元工厂角色

与单纯享元模式中的享元工厂角色一样，复合享元模式中的享元工厂角色也可以按照单例模式实现。下图所示就是将享元工厂角色改写后的结构图。



使用单例模式将享元工厂角色改写后，享元工厂角色的源代码如代码清单 14 所示。

代码清单 14：将一个共享工厂角色用单例模式实现

```
package com.javapatterns.flyweight.composite;
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class FlyweightFactorySingleton
{
    private static FlyweightFactorySingleton myself =
        new FlyweightFactorySingleton();
    private HashMap flies = new HashMap();
    private Flyweight lnkFlyweight;
  /**
   *  构造子是私有的，外界无法直接实例化
   */
    private FlyweightFactorySingleton(){ }
  /**
   *  静态工厂方法，向外界提供惟一的实例
   */
    public static FlyweightFactorySingleton getInstance()
    {
        return new FlyweightFactorySingleton();
    }
  /**
   *  工厂方法，向外界提供含有指定内蕴状态
   *  的复合享元对象
   */
    public Flyweight factory(String complexState)
    {
        // 创建复合享元对象
        ConcreteCompositeFlyweight complexFly =
            new ConcreteCompositeFlyweight();

        int length = complexState.length();
        Character state = null;
        // 设定复合享元对象的内部成分(也就是内部的享元)
        for(int i = 0; i < length; i++)
        {
            state = new Character(complexState.charAt(i));
            System.out.println("factory(" + state + ")");
            complexFly.add( state, this.factory(state));
        }
        return complexFly;
    }
  /**
   *  工厂方法，向外界提供含有指定内蕴状态的单纯享元对象
   */
```

```
    public synchronized Flyweight factory(Character state)
    {
        // 检查具有此状态的享元是否已经存在
        if ( flies.containsKey( state ) )
        {
            // 具有此状态的享元对象已经存在，因此直接返还此对象
            return (Flyweight) flies.get( state );
        }
        else
        {
            // 具有此状态的享元对象不存在，因此创建新对象
            Flyweight fly = new ConcreteFlyweight( state );
            // 将新对象存储到聚集里
            flies.put( state , fly);
            // 将对象返还
            return fly;
        }
    }

    /**
     *  辅助方法，打印出所有已经创建的享元对象清单
     */
    public void checkFlyweight()
    {
        Flyweight fly ;
        int i = 0 ;
        System.out.println("\n======checkFlyweight()========");
        for ( Iterator it = flies.entrySet().iterator() ;
            it.hasNext() ;  )
        {
            Map.Entry e = (Map.Entry) it.next();
            System.out.println( "Item " + (++i) + " : "
                + e.getKey());
        }
        System.out.println("\n======checkFlyweight()========");
    }
}
```

下面给出一个示意性客户端类的源代码，如代码清单 15 所示，显示出调用单例工厂对象的 getInstance()方法，以得到复合享元对象。

代码清单 15：调用本身是单例模式的共享工厂对象的源代码

```
package com.javapatterns.flyweight.composite;
public class ClientSingleton
{
    private static FlyweightFactorySingleton factory;
    public static void main(String[] args)
```



```
{
    // 调用享元工厂的工厂方法得到享元工厂的实例
    factory = FlyweightFactorySingleton.getInstance();
   Flyweight fly;
    // 向享元工厂对象请求一个状态为"aba"的复合享元对象
    fly = factory.factory("aba");
    // 将一个外蕴状态传入到享元对象中
    fly.operation("Composite Call");

    // 调用享元工厂的辅助方法
    factory.checkFlyweight();
}
}
```

可以看到，客户对象向享元工厂申请了一个内蕴状态为“aba”，外蕴状态为“Composite Call”的复合享元对象。

如果调用享元工厂的辅助方法 factory.checkFlyweight()，可以看到系统一共创建了两个单纯享元对象，分别对应于“a”和“b”两种内蕴状态。

31.5　一个咖啡摊的例子

系统的要求

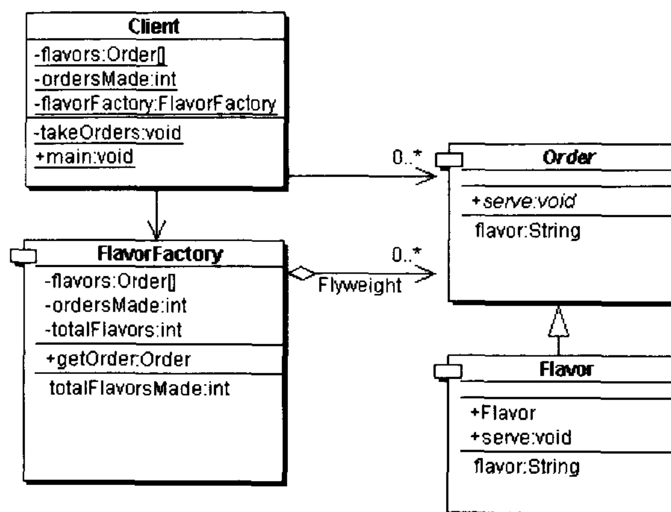
在这个咖啡摊（Coffee Stall）所使用的系统里，有一系列的咖啡“风味（Flavor)”。客人到摊位上购买咖啡，所有的咖啡均放在台子上，客人自己拿到咖啡后就离开摊位。咖啡有内蕴状态，也就是咖啡的风味；咖啡没有环境因素，也就是说没有外蕴状态。

如果系统为每一杯咖啡都创建一个独立的对象的话，那么就需要创建出很多的细小对象来。这样就不如把咖啡按照种类（即“风味”）划分，每一种风味的咖啡只创建一个对象，并实行共享。

使用咖啡摊主的语言来讲，所有的咖啡都可按“风味”划分成如 Capucino、Espresso 等，每一种风味的咖啡不论卖出多少杯，都是全同、不可分辨的。所谓共享，就是咖啡风味的共享，制造方法的共享等。因此，享元模式对咖啡摊来说，就意味着不需要为每一份单独调制。摊主可以在需要时，一次性地调制出足够一天出售的某一种风味的咖啡。

系统的设计

很显然，这里适合使用单纯享元模式，下图所示就是系统的设计图。



在这里享元模式所涉及的角色如下：

- 抽象享元角色：此角色由 Order 类扮演，它是所有的具体享元类的超类，为这些类规定出需要实现的公共接口。
- 具体享元（“风味”）角色：这个角色由 Flavor 类扮演，它的实例是可以共享的。每一个不同的 Flavor 都对应于一个独立而且惟一的享元对象。一个“风味”对象在创建出来之后，其内部状态就不再改变。
- 享元工厂（“风味工厂”）角色：这个角色由 FlavorFactory 类扮演，它负责创建所有的“风味”对象。在客户端提出“风味”对象的请求后，“风味”工厂就会检查是否已经有一个对应的“风味”对象。如果系统里已有这样一个对象存在，就直接返回这个已有的对象；反之，就创建一个新的“风味”对象，并提供给客户端。本角色代表咖啡摊专门调制咖啡的工作人员。
- 客户端（Client）角色：客户端并不直接创建任何的“风味”对象，而是向“风味”工厂提出请求，由“风味”工厂提供一个相应的对象。本角色代表咖啡摊为客人提供服务的侍者。侍者向负责调制咖啡的工作人员请求咖啡，而调制咖啡的工作细节由“风味”工厂封装。

系统的源代码

下面就是本系统的源代码。首先是抽象享元角色的源代码，如代码清单 16 所示。

代码清单 16：抽象享元角色的源代码

```
package com.javapatterns.flyweight.coffeestall;
public abstract class Order
{
    // 将咖啡卖客人
    public abstract void serve();
    // 返还咖啡的名字
    public abstract String getFlavor();
}
```



“风味”角色的源代码如代码清单 17 所示，它实现了抽象 Order 角色所声明的接口，也就是 serve()方法和 getFlavor()方法。“风味”角色就是（单纯）享元对象，“风味”（也就是 flavor 属性）是 Flavor 对象的内蕴状态。一个享元对象的内蕴状态在对象被创建出来以后就不再改变。

serve()方法是享元角色的商业操作，这个操作没有参量，是因为享元对象没有外蕴状态。

代码清单 17：具体享元角色的源代码

```
package com.javapatterns.flyweight.coffeestall;
public class Flavor extends Order
{
    private String flavor;

    /**
     * 构造子，内蕴状态以参量方式传入
     */
    public Flavor(String flavor)
    {
        this.flavor = flavor;
    }

    /**
     * 返还咖啡名字
     */
    public String getFlavor()
    {
        return this.flavor;
    }

    /**
     * 将咖啡卖给客人
     */
    public void serve()
    {
        System.out.println("Serving flavor " + flavor );
    }
}
```

所有的“风味”对象都应当由“风味”工厂提供，而不应当由客户端直接创建。下面就是“风味”工厂的源代码，如代码清单 18 所示。

代码清单 18：工厂角色的源代码

```
package com.javapatterns.flyweight.coffeestall;
public class FlavorFactory
{
    private Order[] flavors = new Flavor[10];
```

```
    private int ordersMade = 0;
    private int totalFlavors = 0 ;
   /**
    *  工厂方法，根据所需的风味提供咖啡
    */
    public Order getOrder(String flavorToGet)
    {
         if (ordersMade > 0)
         {
              for (int i = 0; i < ordersMade; i++)
              {
                   if (flavorToGet.equals((flavors[i]).getFlavor()))
                   {
                        return flavors[i];
                   }
              }
         }
         flavors[ordersMade] = new Flavor(flavorToGet);
         totalFlavors++;
         return flavors[ordersMade++];
    }
   /**
    *  辅助方法，返还创建过的风味对象的个数
    */
    public int getTotalFlavorsMade()
    {
          return totalFlavors;
    }
}
```

下面是系统的客户端角色的源代码，如代码清单 19 所示，这个角色实际上代表咖啡摊的工作人员。

代码清单 19：一个示意性的客户端的源代码

```
package com.javapatterns.flyweight.coffeestall;
/**
 *  客户端角色，代表咖啡摊的侍者
 */
public class Client
{
    // 保持卖出的咖啡的总数目
    private static Order[] flavors = new Flavor[20];
    private static int ordersMade = 0;
    private static FlavorFactory flavorFactory;

   /**
    *  静态方法，提供一杯咖啡
```

```
    */
    private static void takeOrders(String aFlavor)
    {
        flavors[ordersMade++] = flavorFactory.getOrder(aFlavor);
    }
    public static void main(String[] args)
    {
        // 创建风味工厂对象
        flavorFactory = new FlavorFactory();

        // 创建一个个咖啡对象
        takeOrders("Black Coffee");
        takeOrders("Capucino");
        takeOrders("Espresso");
        takeOrders("Espresso");
        takeOrders("Capucino");
        takeOrders("Capucino");
        takeOrders("Black Coffee");
        takeOrders("Espresso");
        takeOrders("Capucino");
        takeOrders("Black Coffee");
        takeOrders("Espresso");
        // 将所创建的咖啡对象卖给客人
        for (int i = 0; i < ordersMade; i++)
        {
            flavors[i].serve();
        }
        // 打印出卖出的咖啡总数
        System.out.println("\nTotal teaFlavor objects made: " +
            flavorFactory.getTotalFlavorsMade());
    }
}
```

可以看出，这个咖啡摊为客人准备最多 20 种不同风味的咖啡。虽然上面的客户端对象叫了 11 杯咖啡，但是所有咖啡的风味却只有三种，即 Capucino、Espresso 和 Black Coffee。运行结果，如代码清单 20 所示。

代码清单 20：咖啡摊系统的运行结果

```
Serving flavor Black Coffee
Serving flavor Capucino
Serving flavor Espresso
Serving flavor Espresso
Serving flavor Capucino
Serving flavor Capucino
Serving flavor Black Coffee
Serving flavor Espresso
```

```
Serving flavor Capucino
Serving flavor Black Coffee
Serving flavor Espresso
Total teaFlavor objects made: 3
```

31.6 咖啡屋的例子

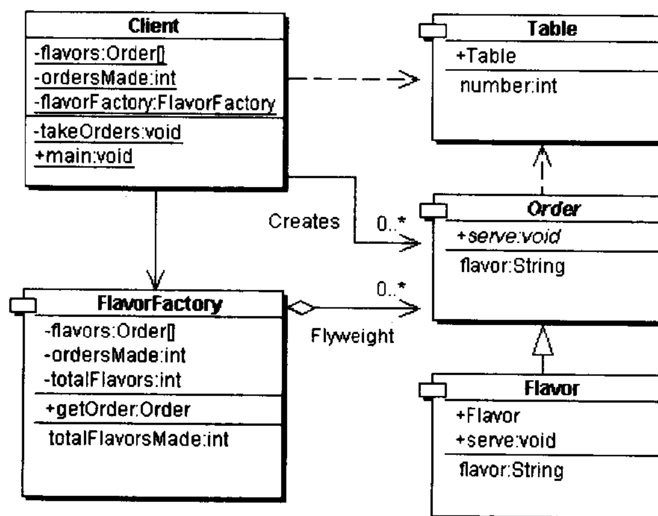
系统的要求

在前面的咖啡摊项目里，由于没有供客人坐的桌子，所有的咖啡均没有环境的影响。换言之，咖啡仅有内蕴状态，也就是咖啡的种类，而没有外蕴状态。

下面考虑一个规模稍稍大一点的咖啡屋（Coffee Shop）项目。屋子里有很多的桌子供客人坐，系统除了需要提供咖啡的“风味”之外，还需要跟踪咖啡被送到哪一个桌位上，因此，咖啡就有了桌子作为外蕴状态。

系统的设计

由于外蕴状态的存在，没有外蕴状态的单纯享元模式不再符合要求。系统的设计可以利用有外蕴状态的单纯享元模式，如下图所示。



在这里享元模式所涉及的角色与咖啡摊项目的角色都是一样的，除了 Table 角色之外：

- 环境角色：由 Table 类扮演，这就是所有的享元角色所涉及的外蕴状态。在本项目中，代表咖啡屋中给客人坐的桌子。每一杯咖啡都以咖啡“风味”作为内蕴状态，这个内蕴状态在享元对象被创建出来之后就不会改变；而“桌子”作为享元对象的外蕴状态，由客户端保存，并在享元对象被创建出来后赋值给享元对象。
- 抽象享元角色：此角色由 Order 类扮演，它是所有的具体享元类的超类，为这些类规定出需要实现的公共接口。



- 具体享元（"风味"）角色：这个角色由 Flavor 类扮演，它的实例是可以共享的。每一个不同的 Flavor 都对应一个独立而且惟一的享元对象。一个"风味"对象在创建出来之后，其内部状态就不再改变。同时，与咖啡摊项目不同的是，本咖啡屋项目的"风味"角色具有外蕴状态，也就是"桌子"对象。
- 享元工厂（"风味工厂"）角色：这个角色由 FlavorFactory 类扮演，它负责创建所有的"风味"对象。在客户端提出请求后，"风味"工厂就会检查是否已经有一个对应的"风味"对象存在，如果已有，就直接返回这个已有的对象；反之就创建一个新的"风味"对象，并提供给客户端。本角色代表咖啡屋专门调制咖啡的工作人员。
- 客户端（Client）角色：客户端并不直接创建任何的"风味"对象，而是向"风味"工厂提出请求，由"风味"工厂提供一个相应的对象。本角色代表咖啡屋的负责招待客人的侍者，侍者代替客人向调制咖啡的工作人员请求咖啡，后者将调制好的咖啡提供给侍者。

系统的源代码

下面就是本系统的源代码。首先是抽象享元角色的源代码，如代码清单 21 所示。

代码清单 21：抽象享元角色的源代码

```
package com.javapatterns.flyweight.coffeeshop;
public abstract class Order
{
   /**
    * 将咖啡卖给客人
    */
    public abstract void serve(Table table);
   /**
    * 返还咖啡名字
    */
    public abstract String getFlavor();
}
```

"风味"角色的源代码如代码清单 22 所示。它实现了抽象 Order 角色所声明的接口，也就是 serve()方法和 getFlavor()方法。"风味"角色就是（单纯）享元对象，"风味"（也就是 flavor 属性）是 Flavor 对象的内蕴状态。一个享元对象的内蕴状态在对象被创建出来以后就不再改变。

代码清单 22：具体享元角色的源代码

```
package com.javapatterns.flyweight.coffeeshop;
public class Flavor extends Order
{
    private String flavor;
   /**
```

```
    *  构造子，内蕴状态以参量方式传入
    */
    public Flavor(String flavor)
    {
        this.flavor = flavor;
    }
   /**
    *  普通方法，返还咖啡名字
    */
    public String getFlavor()
    {
        return this.flavor;
    }
   // 将咖啡卖给客人
    public void serve(Table table)
    {
        System.out.println("Serving table "
            + table.getNumber()
            + " with flavor " + flavor );
    }
}
```

“风味”工厂的源代码如代码清单 23 所示，可以看出这个类与咖啡摊项目并无不同。

代码清单 23：工厂角色的源代码

```
package com.javapatterns.flyweight.coffeeshop;
public class FlavorFactory
{
    private Order[] flavors = new Flavor[10];
    private int ordersMade = 0;
    private int totalFlavors = 0 ;
   /**
    * 工厂方法，根据所需的风味提供咖啡
    */
    public Order getOrder(String flavorToGet)
    {
        if (ordersMade > 0)
        {
            for (int i = 0; i < ordersMade; i++)
            {
                if (flavorToGet.equals((flavors[i]).getFlavor()))
                {
                    return flavors[i];
                }
            }
        }
        flavors[ordersMade] = new Flavor(flavorToGet);
```

```
        totalFlavors++;
        return flavors[ordersMade++];
    }
    /**
     * 辅助方法，返还创建过的风味对象的个数
     */
    public int getTotalFlavorsMade()
    {
        return totalFlavors;
    }
}
```

下面就是咖啡摊项目中没有的一个角色，也就是系统环境角色 Table 类的源代码，如代码清单 24 所示。

代码清单 24：Table 类的源代码

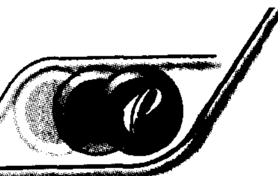
```
package com.javapatterns.flyweight.coffeeshop;
public class Table
{
    // 桌子号码
    private int number;
    /**
     * 构造子
     */
    public Table(int number)
    {
        this.number = number;
    }
    /**
     *  赋值方法
     */
    public void setNumber(int number)
    {
        this.number = number;
    }
    /**
     *  取值方法
     */
    public int getNumber()
    {
        return number;
    }
}
```

下面是系统的客户端角色的源代码，如代码清单 25 所示，与咖啡摊项目的情况一样，这个角色实际上代表咖啡屋的侍者。

代码清单 25：一个示意性的客户端的源代码

```
package com.javapatterns.flyweight.coffeeshop;
public class Client
{
    // 卖出的咖啡总数
    private static Order[] flavors = new Flavor[100];
    private static int ordersMade = 0;
    private static FlavorFactory flavorFactory;
   /**
    * 静态方法，提供一杯咖啡
    */
    private static void takeOrders(String aFlavor)
    {
        flavors[ordersMade++] = flavorFactory.getOrder(aFlavor);
    }
    public static void main(String[] args)
    {
        // 创建风味工厂对象
        flavorFactory = new FlavorFactory();
        // 创建一个个咖啡对象
        takeOrders("Black Coffee");
        takeOrders("Capucino");
        takeOrders("Espresso");
        takeOrders("Espresso");
        takeOrders("Capucino");
        takeOrders("Capucino");
        takeOrders("Black Coffee");
        takeOrders("Espresso");
        takeOrders("Capucino");
        takeOrders("Black Coffee");
        takeOrders("Espresso");
        // 将所创建的咖啡对象卖给客人
        for (int i = 0; i < ordersMade; i++)
        {
            flavors[i].serve(new Table(i));
        }
        // 打印卖出的咖啡总数
        System.out.println("\nTotal teaFlavor objects made: " +
            flavorFactory.getTotalFlavorsMade());
    }
}
```

可以看出，这个咖啡屋提供了 100 种不同风味的咖啡。与咖啡摊项目相同，虽然上面的客户端对象叫了 11 杯咖啡，但是所有的咖啡的风味却只有三种，即 Capucino、Espresso 和 Black Coffee。运行结果如代码清单 26 所示。



代码清单 26：咖啡屋项目的运行结果

```
Serving table 0 with flavor Black Coffee
Serving table 1 with flavor Capucino
Serving table 2 with flavor Espresso
Serving table 3 with flavor Espresso
Serving table 4 with flavor Capucino
Serving table 5 with flavor Capucino
Serving table 6 with flavor Black Coffee
Serving table 7 with flavor Espresso
Serving table 8 with flavor Capucino
Serving table 9 with flavor Black Coffee
Serving table 10 with flavor Espresso
Total teaFlavor objects made: 3
```

31.7　享元模式应当在什么情况下使用

当以下所有的条件都满足时，可以考虑使用享元模式：

（1）一个系统有大量的对象。

（2）这些对象耗费大量的内存。

（3）这些对象的状态中的大部分都可以外部化。

（4）这些对象可以按照内蕴状态分成很多的组，当把外蕴对象从对象中剔除时，每一个组都可以仅用一个对象代替。

（5）软件系统不依赖于这些对象的身份，换言之，这些对象可以是不可分辨的。

满足以上的这些条件的系统可以使用享元对象。

最后，使用享元模式需要维护一个记录了系统已有的所有享元的表，而这需要耗费资源。因此，应当在有足够多的享元实例可供共享时才值得使用享元模式。

31.8　从代码重构的角度看享元模式

本节从代码重构的角度考察一下。虽然享元模式并不是一个非常常见的模式，但是在某些情况下，享元模式可能成为代码重构的强大武器。

问题

在通常情况下，一个初级的 Java 设计师往往会设计一个常规的类。作为一个资深的 Java 设计师，读者可以发现这个系统需要非常多的常规类的实例，而所有这些实例的状态只有非常少的几种。

使用共享策略

根据常识可以推论，这个系统其实并不需要这么多的独立实例，而只需要为每一种不同的状态创建一个实例，让整个系统共享这些很少的实例。换言之，就是使用享元模式。

怎样做到共享

一个享元对象之所以可以被很多的用户端共享，是因为它只含有可以共享的状态，而没有不可以共享的状态，这是可以使用享元模式的重要前提。

因此，问题就转化为怎样才能重新设计这个常规类，使它能够符合享元类的要求。要做到这一点，需要分两步走：

（1）将可以共享的状态和不可以共享的状态从此常规类中区分开来，将不可共享的状态从类里剔除出去。

虽然有很多客户端看上去都在使用此常规类的实例，但是经过仔细考察后读者常常会发现，这些看上去似乎一样的实例其实并非具有完全相同的状态。这时就应当把这些实例的状态分为两类：一类是对所有的客户端都有相同值的状态，另一类是根据不同客户端而有不同值的状态。

那些对所有的客户端都取相同的值的状态是可以共享的状态；而那些对不同的客户端有不同值的状态是不可以共享的状态；那些不可以共享的状态必须从此类中移出。

（2）这个类的创建过程必须由一个工厂对象加以控制。

为了达到共享的目的，客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。这个工厂对象应当使用一个内部列表保存所有的已经创建出来的对象。当客户端请求一个新的对象时，工厂对象首先检查列表，看是否已经有一个对象。如果已经有了，就直接返还此对象；如果没有，就创建一个新的对象。

状态外部化

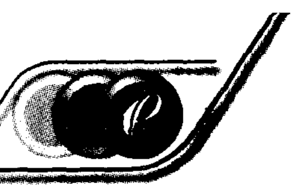
将状态移出享元对象后，这些行为仍然需要这些状态，而大多数情况下在这些状态剥离之后根本就不能正常工作。

解决这个问题的办法有两种：一种是将不能离开这些不能共享的状态的行为一起移到客户端；另一种是由客户端提供这些不能共享的状态。

第一种办法中，所谓将不能剥离的行为与状态一同移出到客户端的做法有明显的缺点。首先，如果有多个客户端的话，会造成同样的逻辑在所有的客户端中重复出现，这不利于代码的复用和维护。其次，将行为移到客户端的做法是破坏封装的原则的。

因此，第二种办法比较好。要使用这种办法就要改写所涉及的方法的参数表，使得客户端可以通过传入恰当的参数将所需要的状态传进去。

换言之，享元模式要求将可以共享的状态设置为内蕴状态，而将不可以共享的状态设置成外蕴状态，将它们外部化。比如，在咖啡屋项目中，客人使用的桌子是一个外蕴状态，可以从咖啡卖单中剔除。而一旦将“桌子”剔除后，咖啡卖单就变成仅有咖啡“风味”状



态了，因此，可以用 Flavor 对象以共享的方式提供给客户端。客户端在“风味”对象被创建出来之后，将“桌子”状态传递给“风味”对象。换言之，侍者在取得咖啡时，会将桌子的号码标记到咖啡上。

创建过程的控制

应用享元模式的另一个关键便是享元对象的创建过程必须由一个工厂对象加以控制。通常情况下都是使用 new 关键字来创建一个类的实例，而在这里则必须使用工厂方法模式来达到创建对象的目的。

在按照以上的过程对代码进行重构之后，便可以达到应用享元模式优化资源的目的。

31.9　享元模式的优点和缺点

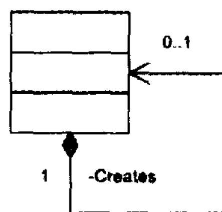
享元模式的优点在于它大幅度地降低内存中对象的数量。但是，它做到这一点所付出的代价也是很高的：

- 享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。
- 享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

31.10　相关的模式

享元模式与单例模式的关系

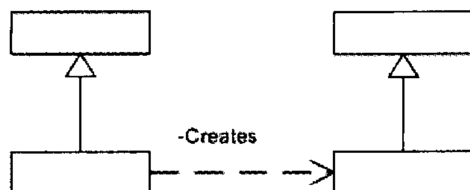
享元模式中的享元对象常常与单例模式中的单例对象相混淆。单例模式的简略类图如右图所示。



单例模式只有一个实例。有一些初学者认为，享元模式中的单纯享元对象相对于每一个内蕴状态而言是惟一的，这意味着它们是单例对象，这是错误的。一个单例类是不能直接实例化的，而单纯享元类是可以直接实例化的。

享元工厂与工厂方法模式

享元工厂角色是一个特殊的工厂方法模式，特殊之处在于这个工厂维护一个所创建过的产品对象的记录，并根据这个记录和产品对象的内部状态循环使用这些产品对象。工厂方法模式的简略类图如下图所示。

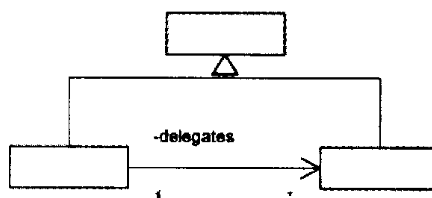


享元工厂与单例模式

享元工厂（不是享元）往往是单例模式。由于只有一个系列的享元，因此系统只需要一个享元工厂实例，所以享元工厂可以设计成为单例模式。

享元模式与代理模式

尽管享元模式与代理模式是完全不同的模式，它们仍然被初学者混淆。代理模式的简略类图如右图所示。



一些初学者往往认为享元可以像代理对象一样，作为被代理对象的替代物使用，这是不对的。

在代理模式里面，为了避免加载大量的耗费资源的对象，可以使用一些轻量级的代理对象，只在系统真正使用到原始对象时才真正加载它们。

享元模式不是这样。在一个使用享元模式的系统里可能看上去有很多的享元对象，好像都是独立的拷贝，但是它其实是被多个客户端共享的单一的实例。

不变模式

不变对象的内蕴状态在对象被创建之后就不再改变。显然，由于不变对象的状态是与环境彻底无关的，因此只要系统允许，享元对象可以用不变模式实现。

但是，享元对象不一定是不变对象。对享元对象的内蕴状态的要求是不可与环境有关，因为享元对象必须可以共享。这就意味着：第一，如果享元对象具有某个可变的状态，但是此状态与环境无关，那么就不影响享元对象的共享，因此是允许的；第二，如果所有享元对象的状态是同步变化的（比如所有的享元都有一个内蕴状态存储当前的时钟时间），就不会影响享元对象的共享，因此也是允许的。

换言之，不变模式要求一个对象的内部状态在创建出来后就不再变化；而享元模式仅要求享元对象的内蕴状态变化不影响共享，这两者是有很大的区别的。不变模式对一个对象来说是一个很强的要求，而享元模式对享元对象的约束却弱很多。

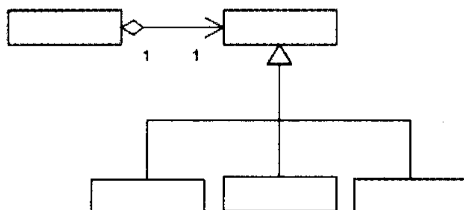
备忘录模式

享元工厂对象维护一个表，通过这个表把很多全同的实例与代表它们的一个对象联系

起来，这就是备忘录模式。

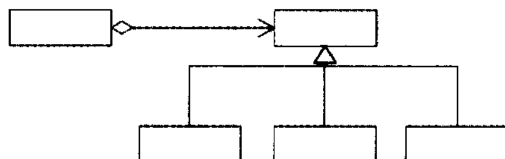
状态模式

状态模式会产生大量的细粒度状态对象，设计师应当认真考虑使用享元模式设计这些状态对象。状态模式的简略类图如下图所示。



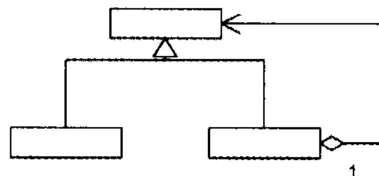
策略模式

策略模式的抽象策略角色是比较重的，而其余的具体策略角色则越轻越好。如果系统需要大量的具体策略角色的实例，那么设计师应当认真考虑使用享元模式设计这些具体策略对象。策略模式的简略类图如下图所示。



合成模式

严格地讲，享元模式（Flyweight Pattern）并不是一个单纯的模式，而是一个由数个模式组合而成的复合模式。如前所述，享元模式的工厂角色是一个工厂方法模式，只是工厂的内部记录了所创建的产品实例，并选择使用这些实例。享元对象是合成模式的应用，抽象享元角色就是复合构件角色，而具体享元角色就是具体构件角色。复合享元是树枝构件，而单纯享元是树叶构件。合成模式的简略类图如右图所示。



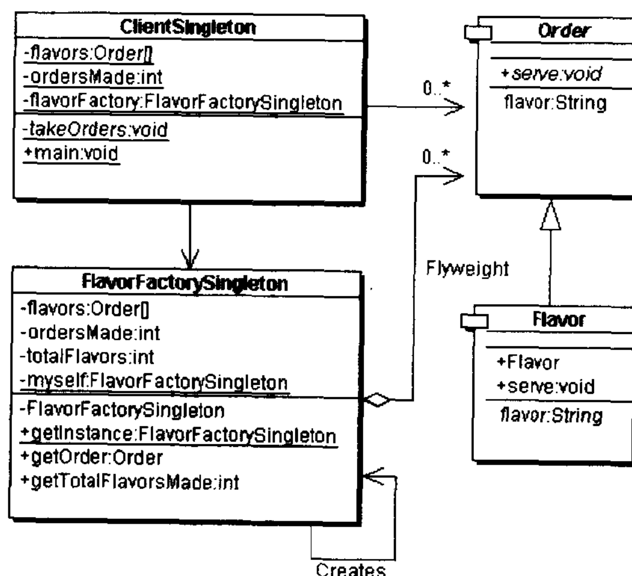
问答题

1. 享元模式要求所有的享元对象都是可以共享的，是这样吗？
2. 请使用单例模式实现“咖啡摊”项目中的“风味工厂”类。
3. 请使用单例模式实现“咖啡屋”项目中的“风味工厂”类。
4. 请问下面的描述是什么模式的应用：“做一颗螺丝钉，服从工作安排”。

问答题答案

1．不是。享元模式允许对享元对象的共享，但是并不要求享元对象都是共享的。比如在复合享元模式中，复合享元对象就是不可以共享的。

2．使用单例模式实现的“咖啡摊”项目的工厂角色的，设计类图如下图所示。



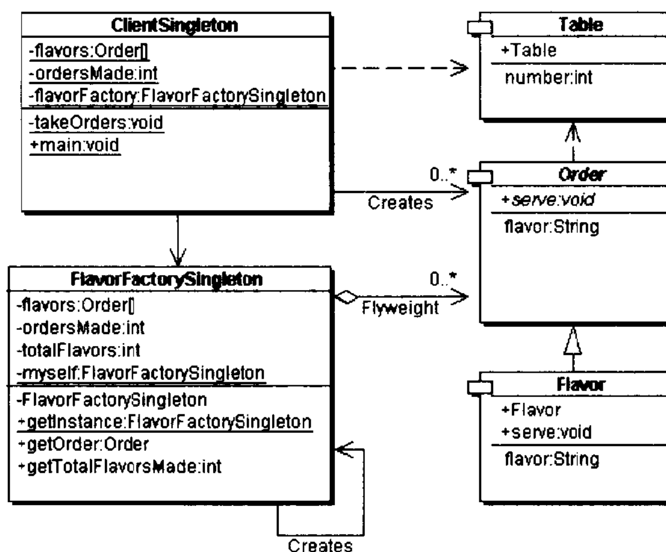
改造之后的“风味”工厂角色的源代码如代码清单 27 所示。

代码清单 27：“风味”工厂角色的源代码

```
package com.javapatterns.flyweight.coffeestall;
public class FlavorFactorySingleton
{
    private Order[] flavors = new Flavor[10];
    private int ordersMade = 0;
    private int totalFlavors = 0 ;
    private static FlavorFactorySingleton myself =
        new FlavorFactorySingleton();
   /**
    *  构造子是私有的，外界无法直接实例化
    */
   private FlavorFactorySingleton(){ }
   /**
    *  静态工厂方法，向外界提供惟一的实例
    */
   public static FlavorFactorySingleton getInstance()
   {
       return myself;
   }
   public Order getOrder(String flavorToGet)
```

```
    {
        if (ordersMade > 0)
        {
            for (int i = 0; i < ordersMade; i++)
            {
                if (flavorToGet.equals((flavors[i]).getFlavor()))
                {
                    return flavors[i];
                }
            }
        }
        flavors[ordersMade] = new Flavor(flavorToGet);
        totalFlavors++;
        return flavors[ordersMade++];
    }
    public int getTotalFlavorsMade()
    {
        return totalFlavors;
    }
}
```

3. 咖啡屋项目的设计图如下图所示。



使用单例模式改写后的“风味”工厂角色的源代码如代码清单 28 所示。

代码清单 28：使用单例模式改写后的咖啡屋系统的源代码

```
package com.javapatterns.flyweight.coffeeshop;
public class FlavorFactorySingleton
{
    private Order[] flavors = new Flavor[10];
    private int ordersMade = 0;
    private int totalFlavors = 0 ;
```

```
        private static FlavorFactorySingleton myself =
            new FlavorFactorySingleton() ;
      /**
        *  构造子是私有的，外界无法直接实例化
        */
        private FlavorFactorySingleton() { }
      /**
        *  静态工厂方法，向外界提供惟一的实例
        */
        public static FlavorFactorySingleton getInstance()
        {
            return myself;
        }
        public Order getOrder(String flavorToGet)
        {
            if (ordersMade > 0)
            {
                for (int i = 0; i < ordersMade; i++)
                {
                    if (flavorToGet.equals((flavors[i]).getFlavor()))
                    {
                        return flavors[i];
                    }
                }
            }
            flavors[ordersMade] = new Flavor(flavorToGet);
            totalFlavors++;
            return flavors[ordersMade++];
        }
        public int getTotalFlavorsMade()
        {
            return totalFlavors;
        }
    }
```

4．这是享元模式的应用。一颗螺丝钉没有环境的影响，可以做到完全共享，这就是享元模式。

参考文献

[JZ91] R. E. Johnson and J. Zweig. Delegation in C++. Journal of Object-Oriented Programming，4(11):22-35，November 1991

[GOF95Z] E.Gamma，R. Helm，R. Johnson，and J. Vlissides. 设计模式——可复用面向对象软件的基础. 李英军、马晓星、蔡敏、刘建中等译. 吕建审校. 机械工业出版社，2000

第 32 章　门面（Facade）模式

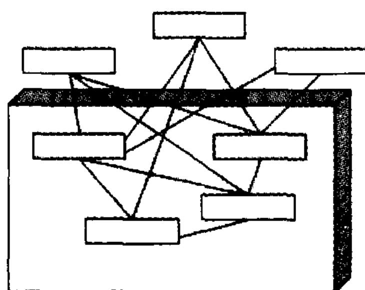
门面模式是对象的结构模式[GOF95]。外部与一个子系统的通信必须通过一个统一的门面（Facade）对象进行，这就是门面模式。

32.1　引　　言

子系统的客户端

现代的软件系统都是比较复杂的，设计模式的任务就是协助设计师处理复杂系统的设计。

设计师处理复杂系统的一个常见方法便是将其"分而治之"，把一个系统划分为几个较小的子系统。但是这样做了以后，设计师往往仍然会发现一个子系统内仍然有太多的类要处理。而使用一个子系统的客户端往往只关注一些特定的功能，需要同时与子系统内部的许多对象打交道后才能达到目的，其对象图如下图所示。



上图描述的是一个客户端必须与许多对象打交道才能完成一个功能，图中的大方框代表一个子系统。

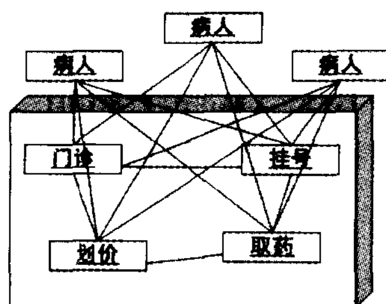
这就是一种不便，它使得系统的逻辑变得不必要的复杂，维护成本提高，复用率降低。

医院的例子

用一个例子进行说明，如果把中国大陆的医院作为一个子系统，按照部门职能，这个系统可以划分为挂号、门诊、划价、化验、收费、取药等。看病的病人要与这些部门打交道，就如同一个子系统的客户端与一个子系统的各个类打交道一样，不是一件容易的事情。

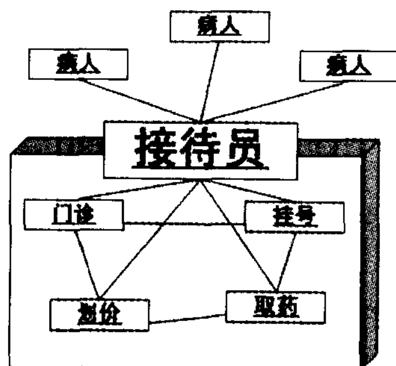
首先病人必须先挂号，然后门诊。如果医生要求化验，病人必须首先划价，然后缴款，才能到化验部门做化验。化验后，再回到门诊室，如下图所示。





上图描述的是病人在医院里的体验，图中的方框代表医院。

解决这种不便的方法便是引进门面模式。仍然通过医院的例子说明，可以设置一个接待员的位置，由接待员负责代为挂号、划价、缴费、取药等。这个接待员就是门面模式的体现，病人只接触接待员，由接待员负责与医院的各个部门打交道，如下图所示。

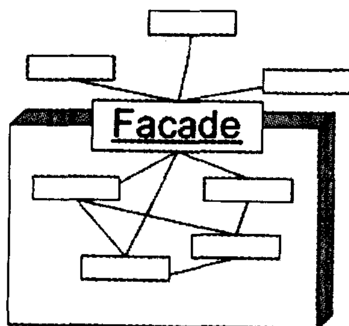


上图描述的是经过门面模式的改装后，病人在医院里的体验，图中的方框代表医院。

什么是门面模式

门面模式要求一个子系统的外部与其内部的通信必须通过一个统一的门面（Facade）对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。

使用了门面模式之后，本章的第一个图中所描述的一个子系统的客户端对象所面对的复杂关系就可以得到简化，如下图所示。



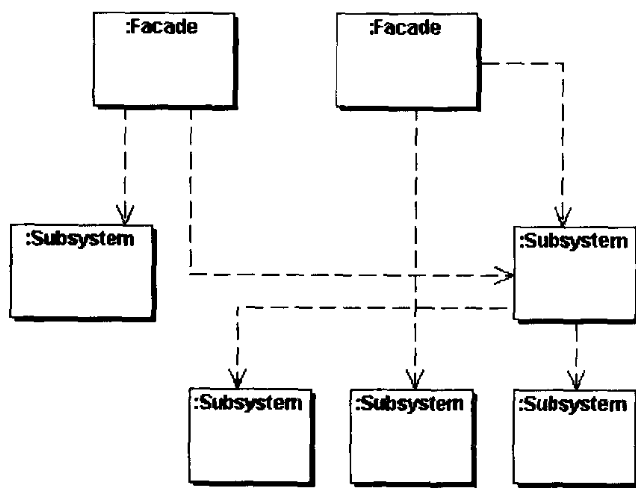
上图描述的是经过门面模式的改装后，一个子系统的客户端与子系统的关系，图中的

大方框代表一个子系统。

就如同医院的接待员一样，门面模式的门面类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与门面对象打交道，而不需要与子系统内部的很多对象打交道。

32.2　门面模式的结构

门面模式是对象的结构模式。门面模式[GOF95]没有一个一般化的类图描述，最好的描述方法实际上就是以一个例子说明。下图所示就是一个门面模式的示意性对象图。



在这个对象图中，出现了两个角色：

- 门面（Facade）角色：客户端可以调用这个角色的方法。此角色知晓相关的（一个或者多个）子系统的功能和责任。在正常情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去。
- 子系统（Subsystem）角色：可以同时有一个或者多个子系统。每一个子系统都不是一个单独的类，而是一个类的集合。每一个子系统都可以被客户端直接调用，或者被门面角色调用。子系统并不知道门面的存在，对于子系统而言，门面仅仅是另外一个客户端而已。

32.3　门面模式的实现

一个系统可以有几个门面类

[GOF]的书中指出：在门面模式中，通常只需要一个门面类，并且此门面类只有一个实例，换言之它是一个单例类。当然这并不意味着在整个系统里只能有一个门面类，而仅仅是说对每一个子系统只有一个门面类。或者说，如果一个系统有好几个子系统的话，每

一个子系统有一个门面类，整个系统可以有数个门面类。

以医院为例，可以为每一个不同的科室配备不同的接待员类。

为子系统增加新行为

初学者往往以为通过继承一个门面类便可在子系统中加入新的行为，这是错误的。门面模式的用意是为子系统提供一个集中化和简化的沟通管道，而不能向子系统加入新的行为。

仍以医院为例，接待员并不是医护人员，接待员并不能为病人提供医疗服务。

如果一个门面模式不能将子系统所有的行为提供给外界，那么可以通过修改门面类或者继承门面类的办法，使门面类或其子类能够将子系统的行为提供给外界。但是，如果一个子系统没有某个行为，想通过修改门面类或者继承门面类的办法来提供这个新的行为是错误的。

32.4 在什么情况下使用门面模式

为一个复杂子系统提供一个简单接口

子系统往往因为不断演化而变得越来越复杂，使用门面模式可以使得子系统更具可复用性。Facade 模式可以提供一个简单的默认视图，对大多数用户来说这个视图已经足够用了，而那些需要进一步继承的用户可以越过 Facade 层直接对子系统进行继承。

子系统的独立性

一般而言，子系统和其他的子系统之间、客户端与实现化层之间存在着很大的依赖性。引入 Facade 模式将一个子系统与它的客户端以及其他的子系统分离，可以提高子系统的独立性和可移植性。

层次化结构

在构建一个层次化的系统时，可以使用 Facade 模式定义系统中每一层的入口。如果层与层之间是相互依赖的，则可以限定它们仅通过 Facade 进行通信，从而简化了层与层之间的依赖关系。

32.5 迪米特法则（LoD）

迪米特法则说："只与你直接的朋友们通信"。迪米特法要求每一个对象与其他对象的

相互作用均是短程的，而不是长程的。只要可能，朋友的数目越少越好。换言之，一个对象只应当知道它的直接合作者的接口。关于迪米特法则的详细讲解，请见本书“迪米特法则”一章。

门面模式创造出一个门面对象，将客户端所涉及的属于一个子系统的协作伙伴的数目减到最少，使得客户端与子系统内部的对象的相互作用被门面对象所取代。显然，门面模式就是实现代码重构以便达到迪米特法则要求的一个强有力的武器。

本章开始时所讨论的医院的例子便是一个从不遵守迪米特法则，通过改造达到迪米特法则要求的例子。

32.6 一 个 例 子

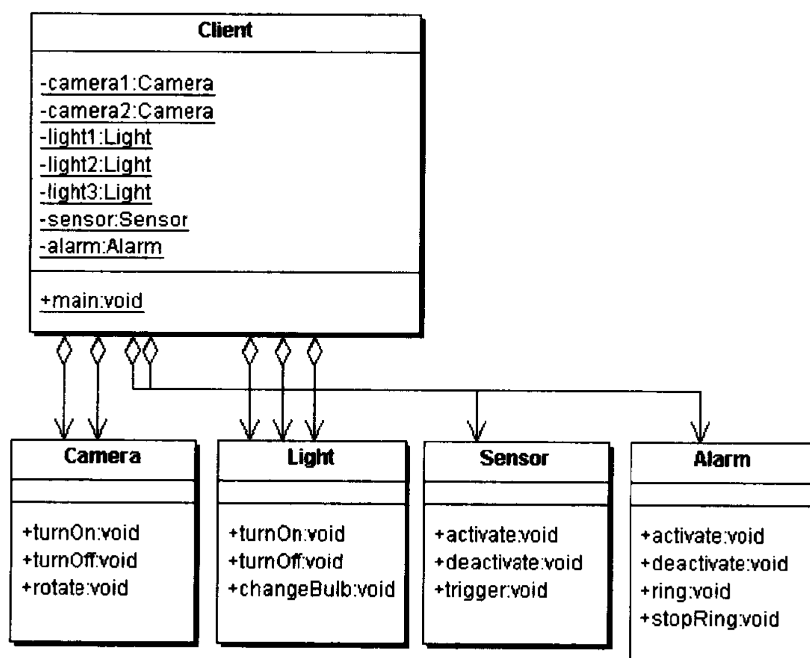
本节拟从代码重构的角度演示门面模式的应用。

本节考察一个保安系统的例子，以说明门面模式的功效。首先给出这个系统不使用门面模式的解决方案，然后指出其缺点，再将门面模式引入，并对源代码进行改组重构。

一个保安系统由两个录像机、三个电灯、一个遥感器和一个警报器组成。保安系统的操作人员需要经常将这些仪器启动和关闭。首先，在不使用门面模式的情况下，操作这个保安系统的操作员必须直接操作所有的这些部件。

不使用门面模式的设计

下图所示就是在不使用门面模式的情况下系统的设计图。



可以看出，Client 对象需要引用到所有的录像机（Camera）、电灯（Light）、感应器

（Sensor）和警报器（Alarm）对象。Client 对象必须对保安系统全知全能，这是一个不利于复用的做法，其源代码如代码清单 1 所示。

代码清单 1：客户端角色 Client 的源代码

```
package com.javapatterns.facade.security;
public class Client
{
    static private Camera camera1, camera2;
    static private Light light1, light2, light3;
    static private Sensor sensor;
    static private Alarm alarm;
    public static void main(String[] args)
    {
        camera1.turnOn();
        camera2.turnOn();
        light1.turnOn();
        light2.turnOn();
        light3.turnOn();
        sensor.activate();
        alarm.activate();
    }
}
```

下面的 Camera、Light、Sensor 和 Alarm 等类均是保安系统的内部类，它们各自提供了不同的方法供客户端调用。Camera 类提供了 turnOn()，turnOff()和 rotate()等方法，如代码清单 2 所示。

代码清单 2：Camera 的源代码

```
package com.javapatterns.facade.security;
public class Camera
{
    //打开录像机
    public void turnOn()
    {
        System.out.println("Turning on the camera.");
    }
    //关闭录像机
    public void turnOff()
    {
        System.out.println("Turning off the camera.");
    }
    //转动录像机
    public void rotate(int degrees)
    {
        System.out.println("rotating the camera by "
            + degrees + " degrees.");
```

```
    }
}
```

Light 类提供了打开 turnOn()，关闭 turnOff()和换灯泡 changeBulb()等方法，如代码清单 3 所示。

代码清单 3：Light 的源代码

```
package com.javapatterns.facade.security;
public class Light
{
    //打开灯
    public void turnOn()
    {
        System.out.println("Turning on the light.");
    }
    //关闭灯
    public void turnOff()
    {
        System.out.println("Turning off the light.");
    }
    //换灯泡
    public void changeBulb()
    {
        System.out.println("changing the light-bulb.");
    }
}
```

感应器 Sensor 类提供了启动 activate()，关闭 deactivate()和触发 trigger()等方法，如代码清单 4 所示。

代码清单 4：Sensor 的源代码

```
package com.javapatterns.facade.security;
public class Sensor
{
    //启动感应器
    public void activate()
    {
        System.out.println("Activating the sensor.");
    }
    //关闭感应器
    public void deactivate()
    {
        System.out.println("Deactivating the sensor.");
    }
    //触发感应器
    public void trigger()
    {
```

```
            System.out.println("The sensor has been triggered.");
        }
    }
```

警报器 Alarm 类提供了启动 activate()、关闭 deactivate()和响铃 ring()等方法，如代码清单 5 所示。

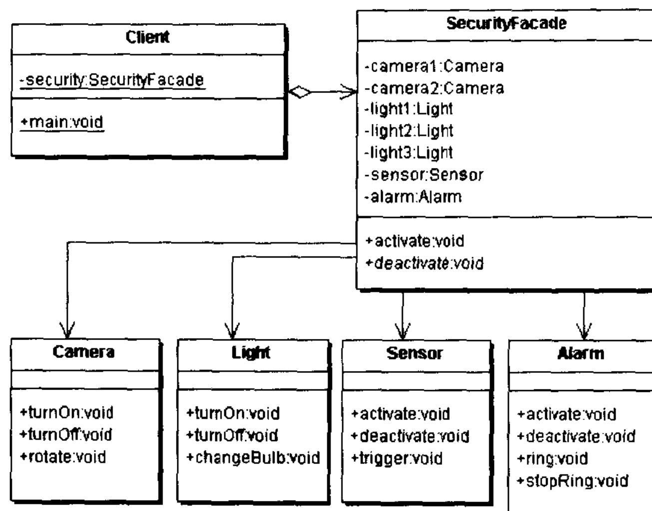
代码清单 5：Alarm 的源代码

```
    package com.javapatterns.facade.security;
    public class Alarm
    {
        //启动警报器
        public void activate()
        {
            System.out.println("Activating the alarm.");
        }
        // 关闭警报器
        public void deactivate()
        {
            System.out.println("Deactivating the alarm.");
        }
        //拉响警报器
        public void ring()
        {
            System.out.println("Ringing the alarm.");
        }
        //停掉警报器
        public void stopRing()
        {
            System.out.println("Stop the alarm.");
        }
    }
```

32.7 使用门面模式的设计

一个合情合理的改进方法就是准备一个系统的控制台，作为保安系统的用户界面。用户只需要操作这个简化的界面就可以操控所有的内部仪器，这实际上就是门面模式的用意。

使用门面模式对前面模拟系统的设计进行改造后，就得到了如下图所示的设计图。



可以看出，门面 SecurityFacade 对象承担了与保安系统内部各个对象打交道的任务，而客户对象只需要与门面对象打交道即可。这是客户端与保安系统之间的一个门户，它使得客户端与子系统之间的关系变得简单和易于管理。

下面是客户端角色的源代码，如代码清单 6 所示。

代码清单 6：客户端 Client 的源代码

```
package com.javapatterns.facade.securityfacade;
public class Client
{
    private static SecurityFacade security;
    public static void main(String[] args)
    {
        security.activate();
    }
}
```

可以看出，客户端程序大大简化了，Client 类只有一个对 SecurityFacade 类的引用。SecurityFacade 类封装了对 Camera、Light、Sensor 和 Alarm 等各个对象的引用，如代码清单 7 所示。

代码清单 7：门面类 SecurityFacade 的源代码

```
package com.javapatterns.facade.securityfacade;
public class SecurityFacade
{
    private Camera camera1, camera2;
    private Light light1, light2, light3;
    private Sensor sensor;
    private Alarm alarm;
    public void activate()
    {
        camera1.turnOn();
```

```
        camera2.turnOn();
        light1.turnOn();
        light2.turnOn();
        light3.turnOn();
        sensor.turnOn();
        alarm.activate();
    }
    public void deactivate()
    {
        camera1.turnOff();
        camera2.turnOff();
        light1.turnOff();
        light2.turnOff();
        light3.turnOff();
        sensor.turnOff();
        alarm.deactivate();
    }
}
```

Camera 类的源代码如代码清单 8 所示。

代码清单 8：Camera 类的源代码

```
package com.javapatterns.facade.securityfacade;
public class Camera
{
    //打开录像机
    public void turnOn()
    {
        System.out.println("Turning on the camera.");
    }
    //关闭录像机
    public void turnOff()
    {
        System.out.println("Turning off the camera.");
    }
    //转动录像机
    public void rotate(int degrees)
    {
        System.out.println("rotating the camera by "
            + degrees + " degrees.");
    }
}
```

Light 类的源代码如代码清单 9 所示。

代码清单 9：Light 源代码

```
package com.javapatterns.facade.securityfacade;
```

```
public class Light
{
    //打开灯
    public void turnOn()
    {
        System.out.println("Turning on the light.");
    }
    //关闭灯
    public void turnOff()
    {
        System.out.println("Turning off the light.");
    }
    //换灯泡
    public void changeBulb()
    {
        System.out.println("changing the light-bulb.");
    }
}
```

感应器 Sensor 类的源代码如代码清单 10 所示。

代码清单 10：Sensor 的源代码

```
package com.javapatterns.facade.securityfacade;
public class Sensor
{
    //启动感应器
    public void activate()
    {
        System.out.println("Activating the sensor.");
    }
    //关闭感应器
    public void deactivate()
    {
        System.out.println("Deactivating the sensor.");
    }
    //触发感应器
    public void trigger()
    {
        System.out.println("The sensor has been triggered.");
    }
}
```

警报器 Alarm 类的源代码如代码清单 11 所示。

代码清单 11：Alarm 的源代码

```
package com.javapatterns.facade.securityfacade;
public class Alarm
```

```
    {
        //启动警报器
        public void activate()
        {
            System.out.println("Activating the alarm.");
        }
        //关闭警报器
        public void deactivate()
        {
            System.out.println("Deactivating the alarm.");
        }
        //拉响警报器
        public void ring()
        {
            System.out.println("Ring the alarm.");
        }
        //停掉警报器
        public void stopRing()
        {
            System.out.println("Stop the alarm.");
        }
    }
```

上面的这个例子得到了文献[YOHTA00]的相关例子的启发，特此注明。

32.8 Session 门面模式

Session 门面模式是 J2EE 模式的一种[J2EE01]，实际上也就是门面模式在 J2EE 框架中的应用。

什么是 Session 门面模式

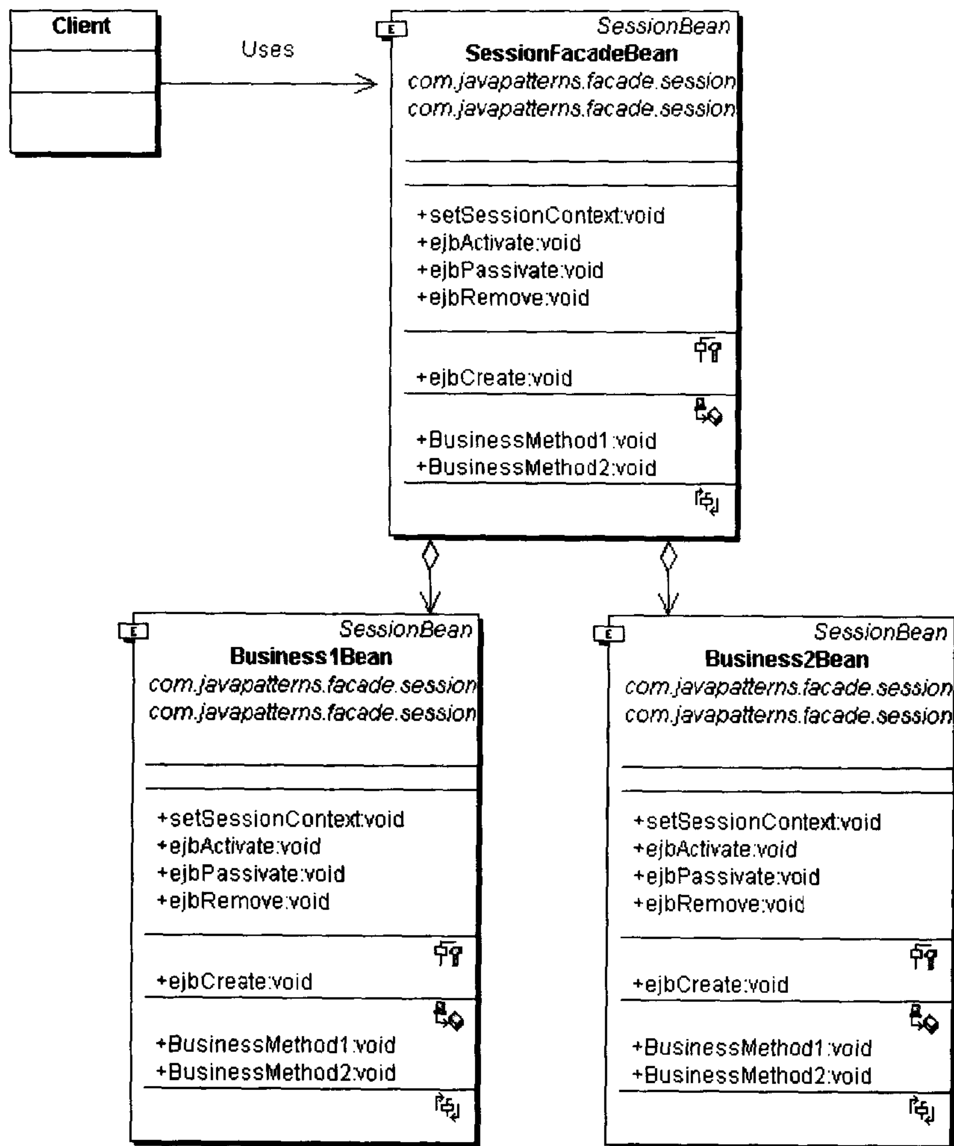
Session 门面模式用一个 SessionBean 作为门面，封装一个工作流程中的商务对象之间的相互作用。Session 门面对象管理商务对象并为客户端提供一个粗粒度的服务层。

Session 门面对象将商务对象之间的相互作用抽象化，它把客户端所需要的接口通过一个服务层暴露给客户端。

Session 门面模式的结构

如下图所示，其中 SessionFacadeBean 就是一个门面类，而 Business1Bean 和 Business2Bean 都是 EntityBean，它们包装了一个子系统的商业逻辑。客户端 Client 类只需要知道这个门面类的接口就可以了，它不需要知道代表商业逻辑的 Business1Bean 和

Business2Bean 的接口。



使用门面模式的语言来描述，Business1Bean 和 Business2Bean 代表一个子系统，而 SessionFacadeBean 将客户端与子系统的内部分隔开，扮演了一个子系统“门面”的作用。

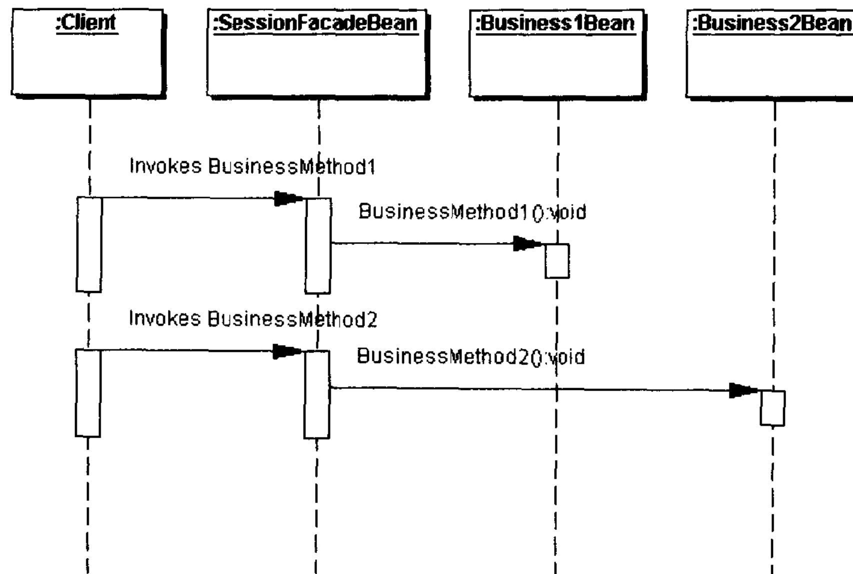
SessionFacadeBean 是一个 SessionBean，用来支持某一个或多个工作流程，它将真正的工作委派给 EntityBean。

Business1Bean 和 Business2Bean 是 EntityBean，实现了工作流程中的一步或几步的商业逻辑。

SessionFacadeBean 知道 Business1Bean 和 Business2Bean 的存在，但是 Business1Bean 和 Business2Bean 并不知道 SessionFacadeBean。在 Business1Bean 和 Business2Bean 看来，SessionFacadeBean 仅仅是一个客户端而已。

而在 Client 对象看来，SessionFacadeBean 就代表了子系统，因为它提供了 Client 对象对这个子系统的全部需求。

Session 门面模式的时序图如下图所示。



参考文献

[DUELL] Michael Duell. Non-Software Examples of Software Design Patterns.

[J2EE01] Deepak Alur，John Crupi，Dan Malks. J2EE Patterns. Best Practices and Design Strategies. by Prentice Hall PTR，2001

[YOHTA00] 太田义则. Skeleton of GOF's Design Pattern. http://www11.u-page.so-net.ne.jp/tk9/ys_oota/mdp/

第 33 章　桥梁（Bridge）模式

桥梁模式虽然不是一个使用频率很高的模式，但是熟悉这个模式对于理解面向对象的设计原则，包括“开–闭”原则（OCP）以及组合/聚合复用原则（CARP）都很有帮助。理解好这两个原则，有助于形成正确的设计思想和培养良好的设计风格。

建议读者将本章与本书的“开–闭原则（OCP）”和“组合/聚合复用原则（CARP）”两章一起阅读。

33.1 引　言

本章就从桥梁模式的用意谈起吧。

桥梁模式的用意

文献[GOF95]在提出桥梁模式的时候指出，桥梁模式的用意是“将抽象化（Abstraction）与实现化（Implementation）脱耦，使得二者可以独立地变化”。这句话很短，但是第一次读到这句话的人很可能都会思考良久而不解其义。

这句话有三个关键词，也就是抽象化、实现化和脱耦。理解这三个词所代表的概念是理解桥梁模式用意的关键。

抽象化

存在于多个实体中的共同的概念性联系，就是抽象化。作为一个过程，抽象化就是忽略一些信息，从而把不同的实体当做同样的实体对待[LISKOV94]。

通常情况下，一组对象如果具有相同的概念性联系，那么它们就可以通过一个共同的类来描述。如果一些类具有相同的概念性联系，往往可以通过一个共同的抽象类来描述。在更加复杂的情况下，可以使用一个继承关系的包括抽象类和具体子类的等级结构来描述。

实现化

抽象化给出的具体实现，就是实现化。

一个类的实例就是这个类的实现化，一个具体子类是它的抽象超类的实现化。而在更加复杂的情况下，实现化也可以是与抽象化等级结构相平行的等级结构，同样可以由抽象类和具体类组成。

脱耦

所谓耦合，就是两个实体的行为的某种强关联。而将它们的强关联去掉，就是耦合的



解脱，或称脱耦。在这里，脱耦是指将抽象化和实现化之间的耦合解脱开，或者说是将它们之间的强关联改换成弱关联。

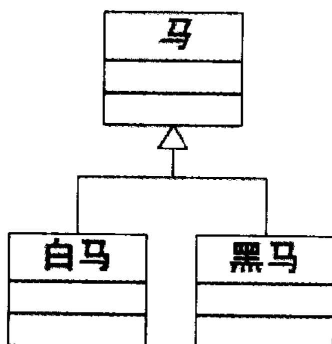
所谓强关联，就是在编译时期已经确定的，无法在运行时期动态改变的关联；所谓弱关联，就是可以动态地确定并且可以在运行时期动态地改变的关联。显然，在 Java 语言中，继承关系是强关联，而聚合关系是弱关联。

将两个角色之间的继承关系改为聚合关系，就是将它们之间的强关联改换成为弱关联。因此，桥梁模式中的所谓脱耦，就是指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使两者可以相对独立地变化。这就是桥梁模式的用意。

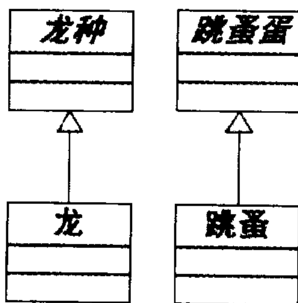
继承关系及其缺点

对数据的抽象化、继承关系、封装和多态性并称 Java 和其他绝大多数的面向对象语言的几大重要特征。

在面向对象的系统设计中，抽象与实现是常见的强有力的武器，也是实现“开–闭原则”的重要武器。一般而言，由 Java 接口或者 Java 抽象类构成抽象角色；而由一个或多个 Java 具体类实现抽象角色所声明的接口，以提供几种不同的实现。比如“马”就是一个抽象角色，而“白马”和“黑马”都是具体角色，如下图所示。



同时，“播下龙种，收获跳蚤”的事情则不可能出现。龙种永远都是龙的种，跳蚤蛋永远都是跳蚤的蛋，如下图所示。



这种看上去似乎是天经地义的套路，其实有不可避免的缺点，即灵活性不够。继承是一种强耦合，它在一开始便把抽象化角色和实现化角色的关系绑定（binding），使得两个

层次之间相互限制，无法独立地演化。

那么，能否使用一种弱耦合来实现抽象化角色和实现者层之间关系的动态绑定呢？换言之，能否允许跳蚤在降生为跳蚤的同时，也有可能降生为龙呢？

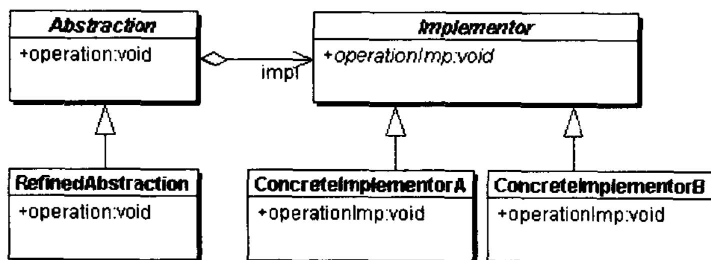
桥梁模式就提供了这样一种用聚合关系实现的弱耦合解决方案。

33.2　桥梁模式的结构

桥梁模式[GOF95]是对象的结构模式，又称为柄体（Handle and Body）模式或接口（Interface）模式。

类图与角色

下图所示就是一个实现了桥梁模式的示意性系统的结构图。



可以看出，这个系统含有两个等级结构，也就是：

- 由抽象化角色和修正抽象化角色组成的抽象化等级结构。
- 由实现化角色和两个具体实现化角色所组成的实现化等级结构。

桥梁模式所涉及的角色有：

- 抽象化（Abstraction）角色：抽象化给出的定义，并保存一个对实现化对象的引用。
- 修正抽象化（Refined Abstraction）角色：扩展抽象化角色，改变和修正父类对抽象化的定义。
- 实现化（Implementor）角色：这个角色给出实现化角色的接口，但不给出具体的实现。必须指出的是，这个接口不一定和抽象化角色的接口定义相同，实际上，这两个接口可以非常不一样。 实现化角色应当只给出底层操作，而抽象化角色应当只给出基于底层操作的更高一层的操作。
- 具体实现化（Concrete Implementor）角色：这个角色给出实现化角色接口的具体实现。

抽象化角色就像是一个水杯的手柄，而实现化角色和具体实现化角色就像是水杯的杯身。手柄控制杯身，这就是此模式别名“柄体”的来源。如果用中国化的语言描述的话，就应当称之为“纲目模式”，而两个等级结构之间则是“纲”与“目”的关系，纲举则目张。

对象是对行为的封装，而行为是由方法实现的。在这个示意性系统里，抽象化等级结构中的类封装了 operation()方法；而实现化等级结构中的类封装的是 operationImp()方法。当然，在实际的系统中往往会有多于一个的方法。

抽象化等级结构中的商业方法通过向对应的实现化对象的委派实现自己的功能，这意味着抽象化角色可以通过向不同的实现化对象委派，来达到动态地转换自己的功能的目的。

源代码

为了便于读者看清楚这个模式是怎样实现的，下面给出这个示意性系统的源代码。首先是抽象化角色的源代码，如代码清单 1 所示。

代码清单 1：抽象化角色的示意性源代码

```
abstract public class Abstraction
{
    protected Implementor imp;
   /**
    *  某个商业方法
    */
    public void operation()
    {
        imp.operationImp();
    }
}
```

显然，在上面给出的抽象化角色是由抽象类 Abstraction 扮演的，它声明了一个商业方法 operation()，并给出了它的实现。



这个实现是通过向实现化对象的委派（调用 operationImp()方法）实现的。

下面则是修正抽象化角色（即子类 RefinedAbstraction）的源代码，如代码清单 2 所示。可以看出，上面讨论过的商业方法被置换掉了。

代码清单 2：修正抽象化角色的示意性源代码

```
public class RefinedAbstraction extends Abstraction
{
   /**
    *  某个商业方法在修正抽象化角色的实现
    */
    public void operation()
    {
        //improved logic
    }
}
```



在实现化角色即抽象类 Implementor 中声明了一个抽象方法 operationImp()，这个方法必须由具体子类实现，如代码清单 3 所示。

代码清单 3：实现化角色的示意性源代码

```
abstract public class Implementor
{
    /**
     *  某个商业方法的实现化声明
     */
    public abstract void operationImp();
}
```

下面是具体实现化角色（抽象类 Implementor）的源代码，如代码清单 4 所示。显然，实现化角色所声明的商业方法 operationImp()被实现了。

代码清单 4：具体实现化角色的示意性源代码

```
public class ConcreteImplementorA extends Implementor
{
    /**
     *  某个商业方法的实现化实现
     */
    public void operationImp()
    {
        System.out.println("Do something...");
    }
}
```

一般而言，实现化角色中的每一个方法都应当有一个抽象化角色中的某一个方法与之相对应，但是，反过来则不一定。换言之，抽象化角色的接口比实现化角色的接口宽。抽象化角色除了提供与实现化角色相关的方法之外，还有可能提供其他的商业方法；而实现化角色则往往仅为实现抽象化角色的相关行为而存在。

33.3　Java 语言中的 Peer 架构

一个 Java 的软件系统总是带有所在的操作系统的视感（Look and Feel）。如果一个软件系统是在 UNIX 系统上面开发的，那么开发人员看到的是 Motif 用户界面的视感；在 Windows 上面使用这个系统的客户看到的是 Windows 用户界面的视感；而一个在 Macintosh 上面使用的用户看到的则是 Macintosh 用户界面的视感。

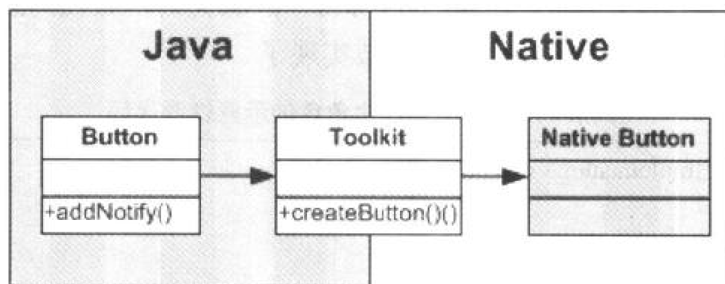
Java 语言是怎样做到这一点的呢？

AWT 的 Peer 架构

Java 语言是通过所谓的 Peer 架构做到这一点的。

在一个 Java 应用程序和真正的显示屏之间还有几个软件架构层次。假设现在考察的是一个 Scrollbar 构件，那么在显示屏上看到的是一个本地的滚动条，与所在的操作系统的任何其他滚动条并无区别。因为这个与系统有关的滚动条实际上是 Java 的 Scrollbar 对象的 Peer 对象所产生的，这个对象的类型是 java.awt.peer.ScrollbarPeer，而不是 Java 应用程序所使用的类型为 java.awt.Scrollbar 的对象。这个 Peer 对象会首先截获诸如鼠标单击这样的事件，然后将有关信息传递给 Java 的 Scrollbar 对象。

一般地，在每一个 Java 构件和它的 Peer 构件之间都有一个 Peer 接口，这个 Peer 接口定义出每一个 Java 构件与其 Peer 构件之间的关系，它使得一个像 Scrollbar 这样的构件可以动态地与任何操作系统的本地滚动条发生 Peer 关系。Button 和 ButtonPeer 的通信图如下图所示，Button 是一个 Java 构件，通过 Toolkit 对象与本地 Peer 构件通信。



一般而言，Java 程序员只需要知道 Java 所提供的构件，而不需要知道系统底层所使用的 Peer 构件，除非是要开发一个新的操作系统的 Peer 构件，或者像本书这样探讨桥梁模式在其中的应用。

Peer 接口

Java 为 AWT 中的每一个 GUI 构件都提供了一个 Peer 构件，这个 Peer 构件是所属的 Java 构件在本地环境中的实现化。比如 Choice 是一个 AWT 提供的 GUI 构件，它允许用户在一个列中选择一个或者多个项。一个 Java 应用程序的开发人员只需要考虑 Choice 对象所提供的接口就可以了。当用户运行这个系统时，这个 Choice 对象被动态地与一个合适的底层 Peer 对象联系起来，这个 Peer 对象会按照指令执行正确的操作。

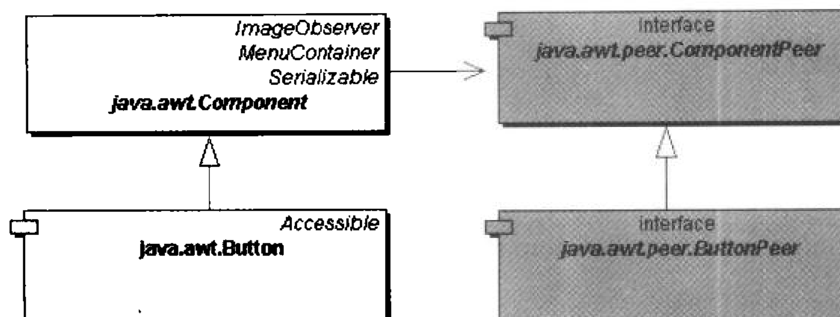
这个将 Java 的 GUI 构件与本地环境的 Peer 构件联系起来的接口，就是所谓的 Peer 接口。一个 Peer 接口其实就是一个定义了 Peer 构件必须实现的各个方法的接口。而 AWT 提供的所有 Peer 接口都放在 java.awt.peer 库中。比如这个库里有一个叫做 ButtonPeer 的接口，这个接口仅含有一个 setLabel()的方法，其源代码如代码清单 5 所示。

代码清单 5：ButtonPeer 的源代码

```
package java.awt.peer;
import java.awt.*;
public interface ButtonPeer extends ComponentPeer
{
    void setLabel(String label);
}
```



这意味着所对应的本地对象必须实现这个方法，才可以成为 AWT 的 Peer 对象。当然，正如 Button 本身还是 Component 的子类型，因而必须实现 Component 所规定出的抽象方法一样，ButtonPeer 本身还是 ComponentPeer 的子类型，所以必须实现 ComponentPeer 接口所规定的各个方法。两种颜色代表两个不同的等级结构，如下图所示。

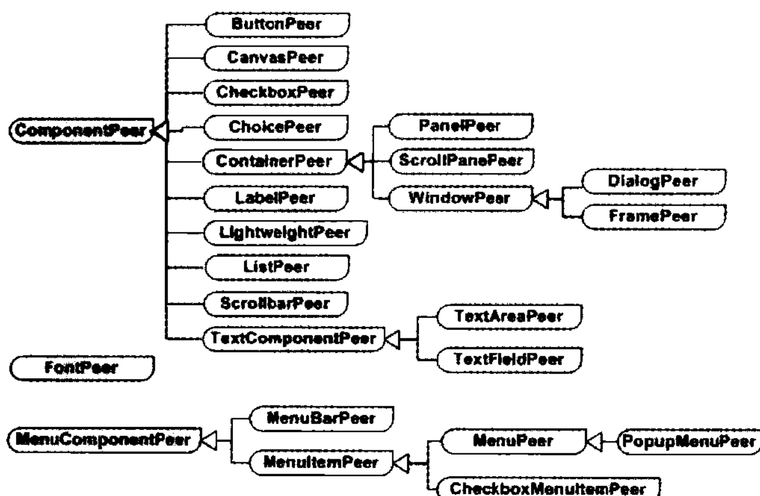


在所有的 Component 类型与 ComponentPeer 类型之间存在着一对一的关系（只有 LightweightPeer 是一个例外），比如 Window 有 WindowPeer，Checkbox 有 CheckboxPeer 等。

一般而言，在 Peer 接口与它们所对应的 Java 构件的接口之间也存在着对应关系。严格地讲，Peer 接口中的每一个方法都对应于 Java 构件中的一个方法。当然，一个 Peer 构件实现了一些不得不实现的方法，并不意味着必须提供有意义的实现，比如一个 Peer 构件完全可以为某一个方法提供一个空的实现。同时，一个 Java 构件可能会具有很多其他的方法，是相应的 Peer 构件所没有的。换言之，Java 构件的接口比相应的 Peer 接口宽。这些多出来的方法往往是与 Peer 和本地无关的、完全由 Java 处理的行为。

java.awt.peer 库

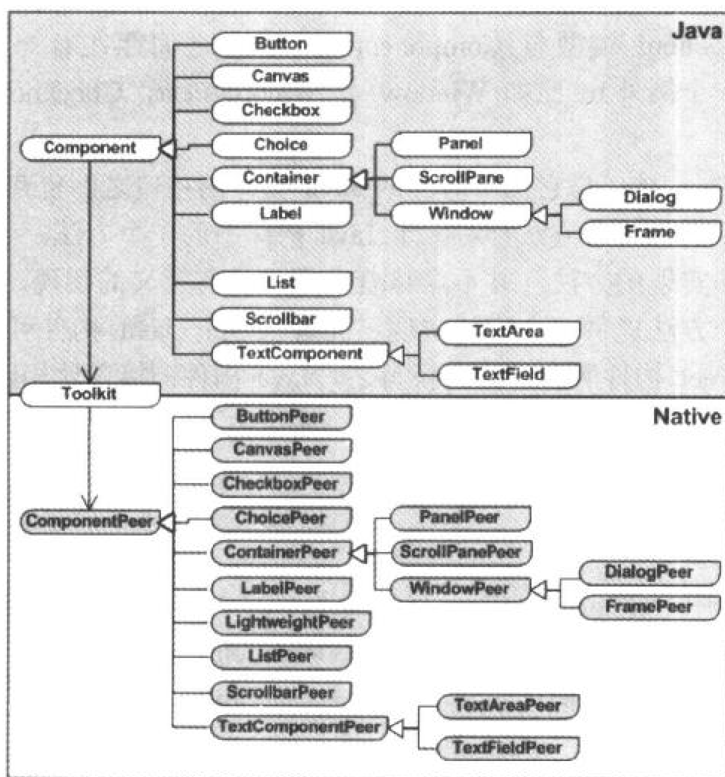
在 java.awt.peer 中，ComponentPeer 接口是所有的（非菜单）视窗构件的 Peer 接口的超接口，成为所有 Peer 构件的超类型。MenuComponentPeer 则给出了所有的菜单视窗构件的超类型，FontPeer 是 Font 构件的超类型，java.awt.peer 库的类图如下图所示。



创建一个视窗构件并不会立即创建一个对应的 Peer 对象，只有当一个视窗构件的 addNotify()方法被调用时，所对应的 Peer 构件才会被创建出来。只有当用户与一个视窗构件发生相互作用时才绝对需要它的 Peer 构件。因此，一个应用系统可以将 Peer 构件的创建推迟到必须创建的时候，以节省资源。

桥梁模式的应用

在 AWT 库中的每一个 Component 的子类都有一个 ComponentPeer 的子类与之匹配。所有 Component 的子类都属于一个等级结构，而所有的 ComponentPeer 的子类都属于另一个等级结构。Component 类型和 ComponentPeer 类型通过 Toolkit 对象相互通信，如下图所示。



从上面的类图可以看出，抽象类 Component 是各种具体的视窗构件的超类型。这些视窗构件是与操作系统的细节无关的，它们向外界提供统一的功能接口。ComponentPeer 是各种 Peer 对象的超类型，这些 Peer 对象负责处理与操作系统细节密切相关的功能的实现。具体视窗构件对象通过向相应的 Peer 对象进行功能委派，达到复用 ComponentPeer 的功能的目的。

从上图中也可以看出，Component 相当于抽象角色，Button 相当于修正抽象角色；ComponentPeer 相当于实现化角色，而 ButtonPeer 相当于具体实现化角色。系统根据当前操作系统动态地选择 Button 对象所使用的底层实现。

Java 的 AWT 就是这样做到“Write once, run anywhere”的。



抽象工厂模式的应用

Toolkit 使用抽象工厂模式创建各种 Peer 对象，请读者参见本书的“抽象工厂（Abstract Factory）模式”一章。

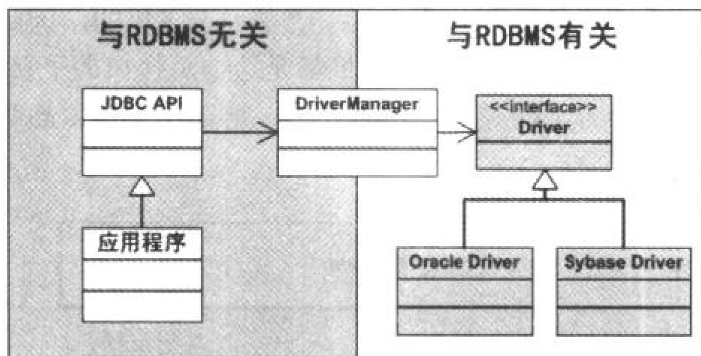
33.4　驱动器和 JDBC 驱动器

大多数的驱动器（Driver）都是桥梁模式的应用[METSKER02]。使用驱动程序的应用系统就是抽象化角色，而驱动器本身扮演实现化角色。

JDBC 驱动器

一个熟知的驱动器的例子就是 JDBC 驱动器。JDBC 为所有的关系数据库提供一个通用的界面。一个应用系统动态地选择一个合适的驱动器，然后通过驱动器向数据库引擎发出指令。这个过程就是将抽象角色的行为委派给实现角色的过程。

抽象角色可以针对任何数据库引擎发出查询指令，因为抽象角色并不直接与数据库引擎打交道，JDBC 驱动器负责这个底层的工作。由于 JDBC 驱动器的存在，应用系统可以不依赖于数据库引擎的细节而独立地演化；同时数据库引擎也可以独立于应用系统的细节而独立演化。两个独立的等级结构如下图所示，左边是 JDBC API 的等级结构，右边是 JDBC 驱动器的等级结构。应用程序是建立在 JDBC API 的基础之上的。



应用系统作为一个等级结构，与 JDBC 驱动器这个等级结构是相对独立的，它们之间没有静态的强关联。应用系统通过委派与 JDBC 驱动器相互作用，这是一个桥梁模式的例子。

其他驱动器

实际上，任何其他的驱动器软件都可以是桥梁模式的应用。比如打印机驱动器将使用打印机的应用系统（比如一个文字处理器）与打印机的驱动细节分割开，使得应用系统和打印机驱动器可以相对独立地演化。

应用系统与打印机驱动器形成两个独立的等级结构，一个文字处理器可以动态选择任何一个打印机驱动器，它们之间通过动态的委派相互作用。一个文字处理器应用系统不必处理直接打印机语言，因为打印机驱动器会做这个工作，这使得应用程序与驱动器可以独立演化。

33.5 从重构的角度考察

所谓重构，就是在不增加或减少一个软件系统的功能的情况下，对代码的结构进行优化。

本节从一般性的设计原则出发，演示如何从一个简单系统的初级设计方案，通过重构发展出具有高复用性的、以桥梁模式为基础的设计方案来。

问题

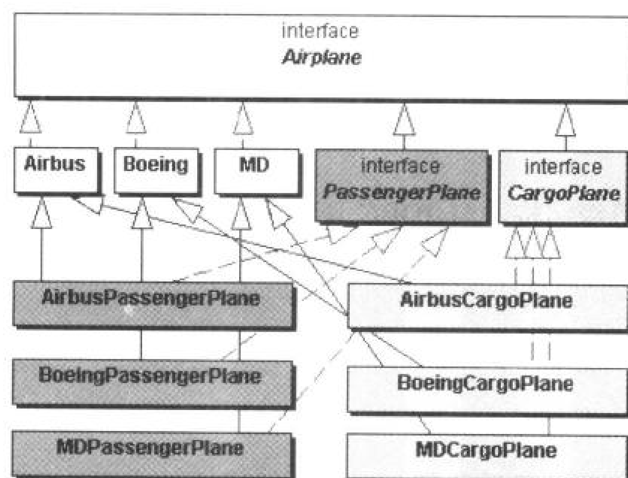
空中巴士（Airbus）、波音（Boeing）和麦道（McDonnell-Douglas）都是飞机制造商，它们都生产载客飞机（Passenger Plane）和载货飞机（Cargo Plane）。现在需要设计一个系统，描述这些飞机制造商以及它们所制造的飞机种类。

设计方案一

系统的设计

这看上去不像是一个困难的问题。首先，系统是关于飞机的，因此可以设计一个总的飞机接口，不妨叫做 Airplane。其他所有的飞机都是这个总接口的子接口或者具体实现。

下图所示就是这个设计方案的类图，可以看出，这是一个不太高明的设计，导致了一团理不清的关系。



可以看出，在这个设计方案里面，出现了两个子接口，分别表示客机和货机。所有的



具体飞机又都要继承自 Airbus，Boeing 和 MD 等超类。这样一来，每一个具体飞机都带有两个超类型：飞机制造商类型，客、货机类型。

“开–闭”原则

仔细思考一下就会发现，这样的设计有一个很大的缺陷：在具体飞机与飞机制造商、飞机种类之间的耦合过强。一旦出现下面的情况，系统的设计就不可避免地需要修改：

- 需要向系统引进新的飞机制造商；
- 需要向系统引进新的飞机类型，

组合/聚合复用原则(CARP)

所谓组合/聚合复用原则讲的是要尽量使用合成/聚合，而不是继承关系来达到扩展系统功能的目的。组合/聚合复用原则是达到“开–闭”原则要求的必要手段。

显然，在这个系统设计里面，过多地使用了继承关系。由于继承关系是一种静态的关联，因此很多涉及到这个关系的变化都会导致代码的修改。

改变这一局面的办法已经由组合/聚合复用原则指出来了，那就是放弃继承关系，通过动态的委派改进系统的设计。

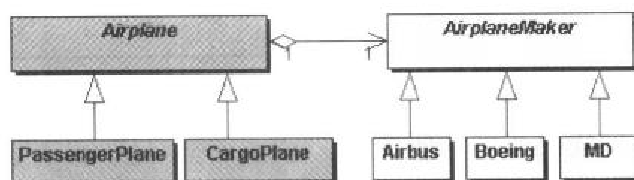
设计方案二

要克服第一个设计方案的缺点，可以使用桥梁模式这一锐利武器。

系统的设计

使用桥梁模式的关键在于准确地找出这个系统的抽象化角色和具体化角色。从系统所面对的问题不难看出，代表飞机的抽象化的是它的类型，也就是“客机”或者“货机”；而代表飞机的实现化的则是飞机制造商。

因此，可以给出一个更有灵活性的设计，这个使用了桥梁模式而做的新设计如下图所示。



在上图的设计中，客机和货机经过一个飞机的“转世”桥梁，可以分别“投胎”到空中巴士、波音和麦道等飞机制造商那里，“出生”为不同牌子的飞机。

由于这个“转世”桥梁实际上是一个聚合关系，因此可以动态地变化。所以如果系统需要加入新的飞机种类或者飞机制造商的话，已有的各个角色不必改变，需要改变的仅仅是一个多态性的聚合关系。

示意性源代码

下面给出这个系统的示意性 Java 源代码。

首先 Airplane 扮演抽象化角色，它声明出所有修正抽象化角色所需的接口，它的源代

码如代码清单 6 所示。

代码清单 6：抽象化角色类 Airplane 接口的源代码

```
abstract public class Airplane
{
    abstract public void fly() ;
    protected AirplaneMaker airplaneMaker;
}
```

载客飞机作为飞机的种类，属于修正抽象化角色。它的示意性源代码如代码清单 7 所示。

代码清单 7：载客飞机的源代码

```
public class PassengerPlane extends Airplane
{
    public void fly()
    {
    //Write your code here
    }
}
```

载货飞机作为飞机的另一个种类，也属于修正抽象化角色。它的示意性源代码如代码清单 8 所示。

代码清单 8：载货飞机的源代码

```
public class CargoPlane extends Airplane
{
    public void fly()
    {
    //Write your code here
    }
}
```

飞机制造商 AirplaneMaker 扮演实现化角色，它给出所有的修正抽象化角色所需要实现的接口。它的源代码如代码清单 9 所示。

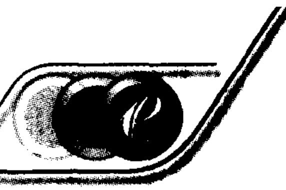
代码清单 9：AirplaneMaker 的示意性源代码

```
abstract public class AirplaneMaker
{
    abstract public void produce();
}
```

空中客车是飞机制造商之一，属于具体实现化角色。这里给出的是它的象征性 Java 源代码，如代码清单 10 所示。

代码清单 10：空中客车的示意性源代码

```
public class Airbus extends AirplaneMaker
```



```
{
    public void produce()
    {
        //Write your code here
    }
}
```

波音是另一个飞机制造商，同样属于具体实现化角色。这里给出的是它的示意性 Java 源代码，如代码清单 11 所示。

代码清单 11：波音的示意性源代码

```
public class Boeing extends AirplaneMaker
{
    public void produce()
    {
        //Write your code here
    }
}
```

同样地，飞机制造商麦道也属于具体实现角色，它的示意性 Java 源代码如代码清单 12 所示。

代码清单 12：飞机制造商麦道的示意性源代码

```
public class MD extends AirplaneMaker
{
    public void produce()
    {
        //Write your code here
    }
}
```

再谈“开–闭”原则（OCP）

现在，如果需要增加新的飞机制造商，或者新的飞机种类的话，只需要向系统引进一个新的修正抽象化角色，或者一个新的具体实现化角色就可以了。或者说，系统的功能可以在不修改已有代码的情况下得到扩展。

换言之，这个设计是符合“开–闭”原则的。

再谈组合/聚合复用原则（CARP）

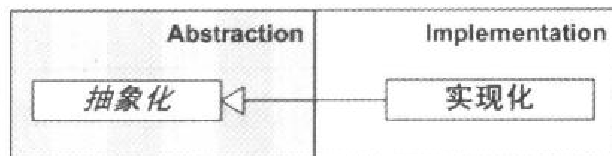
由于这个新的系统设计适当地使用了继承关系来构建两个等级结构内部的结构，并且使用聚合关系来构建两个等级结构之间的结构，因此，这是一个通过使用“组合/聚合复用原则（CARP）”来达到“开–闭”原则要求的范例。

对变化的封装

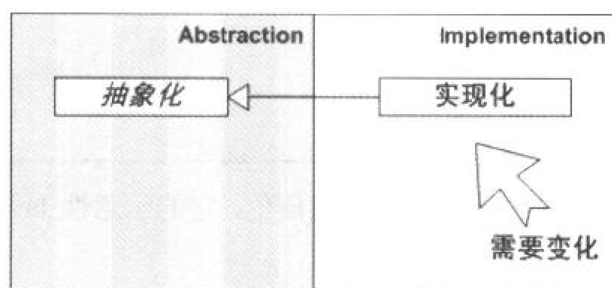
“找到系统的可变因素，将之封装起来”，通常就叫做“对变化的封装”。对变化的封

装实际上是达到“开–闭”原则的途径，与组合/聚合复用原则是相辅相成的。

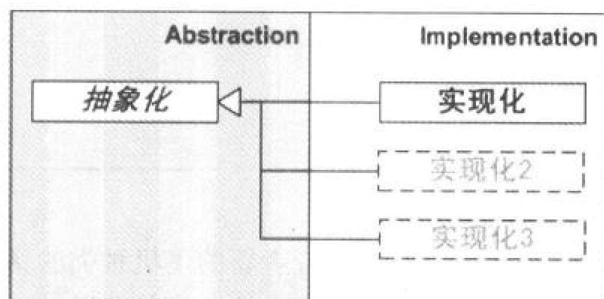
抽象化与实现化的最简单实现，也就是“开－闭”原则在类层次上的最简单实现，如下图所示。



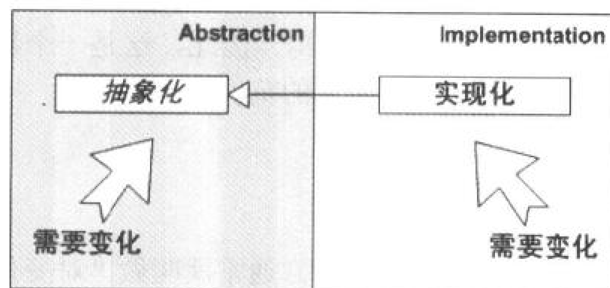
一般来说，一个继承结构中的第一层是抽象角色，封装了抽象的商业逻辑，这是系统中不变的部分。第二层是实现角色，封装了设计中会变化的因素。这个实现允许实现化角色有多态性变化，如下图所示。



换言之，客户端可以持有抽象化类型的对象，而不在意对象的真实类型是“实现化”、“实现化 2”还是“实现化 3”，如下图所示。

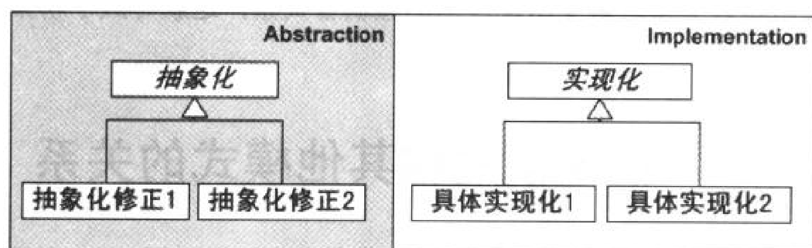


显然，每一个继承关系都封装了一个变化因素，而一个继承关系不应当同时处理两个变化因素。换言之，这种简单实现不能够处理抽象化与实现化都面临变化的情况，如下图所示。

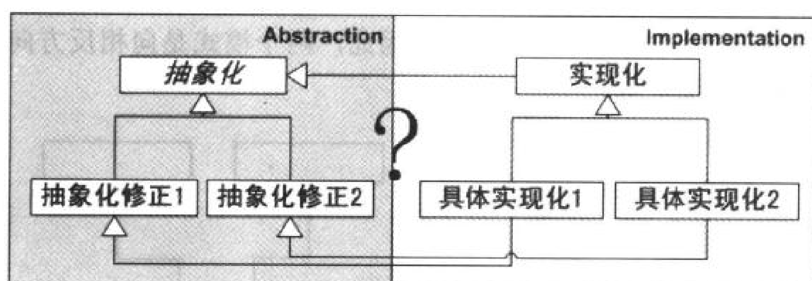




上图中的两个变化因素应当是彼此独立的，可以在不影响另一者的情况下独立演化。比如，下面的两个等级结构分别封装了自己的变化因素，由于每一个变化因素都是可以通过静态关系表达的，因此分别使用继承关系实现，如下图所示。

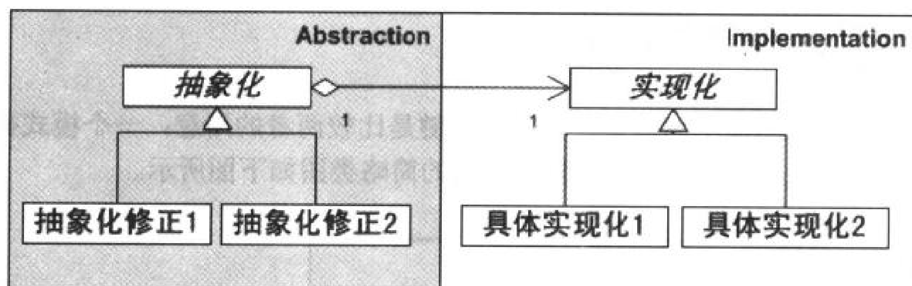


那么在抽象化与实现化之间的变化怎么办呢？好的设计往往只有一个，而不好的设计可以有很多种。下图所示就是一种不恰当地继续沿用继承关系的老路进行静态关系设计的类图。



可以看出，这个实现有很多问题。首先，这种多重继承关系设计能不能在 Java 语言中实现都是问题；其次，如果出现新的抽象化修正角色或者新的具体实现化角色，设计师就只好重新修改现有的系统中的静态关系，以适应新的角色。这就破坏了“开-闭”原则。

正确的设计方案应当是使用两个独立的等级结构封装两个独立的变化因素，并在它们之间使用聚合关系，以达到功能复合的目的。这就自然地将设计引导到桥梁模式上面，如下图所示，抽象化与实现化的变化分别得到封装之后，使用聚合关系联系抽象化角色与实现化角色。



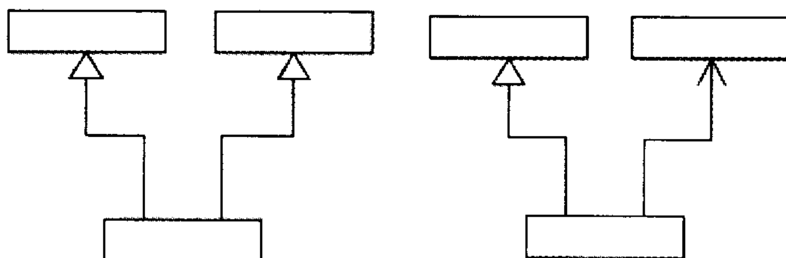
从另一个角度讲，一个好的设计通常没有多于两层的继承等级结构。或者说，如果出现两个以上的变化因素，就需要找出哪一个因素是静态的，可以使用继承关系；哪一个是动态的，必须使用聚合关系。

桥梁模式是“对变化的封装”原则以及组合/聚合复用原则的极好例子。在前面飞机制造系统中，飞机的种类和制造商代表两个不同的变化因素，而这两个变化因素需要独立地变化。按照对变化的封装原则，它们应当被封装到继承的等级结构中。而两个等级结构之间应当选择使用聚合关系，避免出现静态的强耦合关系，这就导致了桥梁模式的设计方案。

33.6 桥梁模式和其他模式的关系

桥梁模式和适配器模式的区别

对象形式的适配器模式可能看上去很像桥梁模式。然而适配器模式的目的是要改变已有的接口，让它们可以相容，以使没有关系的两个类能在一起工作。而桥梁模式是分离抽象化和实现化，使得两者的接口可以不同。 因此，两个模式是向相反方向努力的。适配器模式的简略类图如下图所示。

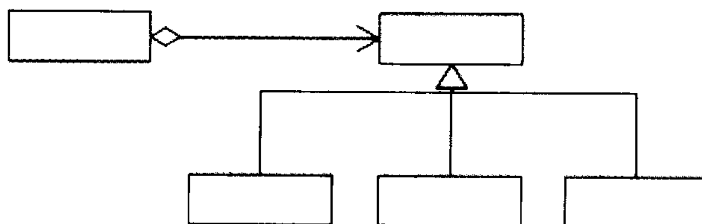


Java 语言的出现在[GOF95]一书出现之前，因此有许多 Java 名词和文档是在设计模式的名词和文档出现之前出现的。这样就造成一些 Java 语言中的名词在模式的字典中有不同解释的情况。比如 Java 语言文档中有一些地方使用“桥梁”一词，仅仅表示一种连接关系，这并不表示与桥梁模式有关。在本章的问答题里读者可以看到一些例子。

适配器模式经常会与桥梁模式一同使用。

桥梁模式和策略模式的区别

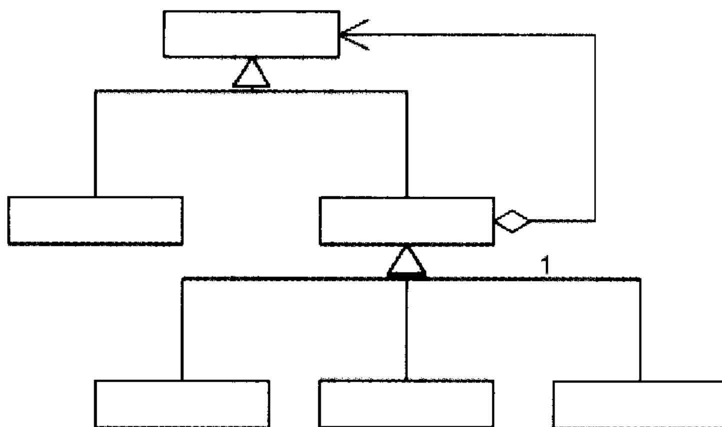
区别类图相似而本质不同的设计模式的关键是比较两者的用意：一个模式被设计出来要解决的问题，便是此模式的用意。策略模式的简略类图如下图所示。



策略模式经常和桥梁模式相混淆。尽管两者的结构看上去很相像，但它们是用来解决完全不同的设计问题的。策略模式是关于计算方法的封装的，而桥梁模式是关于怎样把抽象化角色和实现化角色的强耦合解除掉的。桥梁模式的目的是要为同一个抽象化角色提供不同的实现。

桥梁模式与装饰模式的关系

桥梁模式与装饰模式的用意确有相似之处，它们都要避免太多的子类。如果有两个基类，它们各有很多的子类，那么怎么划分这些类使得客户端需要尽量少的子类呢？装饰模式的简略类图如下图所示。



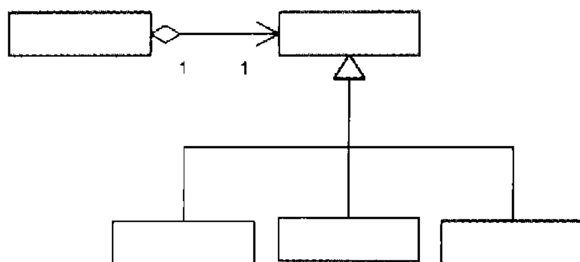
装饰模式的办法是把每个子类中比基类多出来的行为放到单独的类里面。这样当这些描述额外行为的对象被封装到基类对象里面时，就得到了所需要的子类对象。将这些描述额外行为的类，排列组合可以造出很多的功能组合来。如果用静态继承的方法创建这些组合出来的类所涉及到的工作量很大，以致实际上很难做到。装饰模式要求所有的这些“额外行为类”具有相同的接口。关于装饰模式，请读者参见本书的“装饰（Decorator）模式”一章。

桥梁模式的解决办法则又有所不同。桥梁模式把原来的两个基类的实现化细节抽出来，再建造到一个实现化的等级结构中，然后再把原有的基类改造成一个抽象化等级结构。

桥梁模式中抽象化角色里面的子类不能像装饰模式那样嵌套。桥梁模式却可以连续使用。换言之，一个桥梁模式的实现化角色可以成为下一步桥梁模式的抽象化角色。

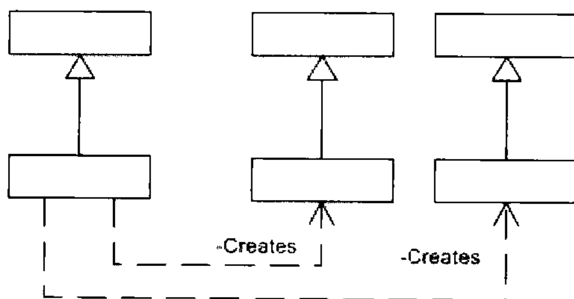
桥梁模式与状态模式的关系

桥梁模式描述两个等级结构之间的关系，状态模式则描述一个对象与其（外部化的）状态对象之间的关系。显然，状态对象可以看做是桥梁模式的一个退化后的特殊情况，而在使用状态模式的系统中，类型与子类型的变化会使得系统向桥梁模式演化[LISKOV94]。状态模式的简略类图如下图所示。



桥梁模式与抽象工厂模式的关系

桥梁模式的设计会涉及到多个等级结构的处理，而抽象工厂模式也要涉及到多个等级结构的问题。抽象工厂模式的简略类图如下图所示。



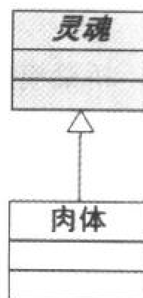
在抽象工厂模式中，产品可以按照等级结构和产品族来分类。等级结构是静态的关系，处于同一个等级结构中的产品之间的关系是静态的关系；而产品族是动态的关系，系统使用哪一个产品族的产品是可以在运行时期决定的。这就是两个模式的相似之处。

抽象工厂模式中的工厂等级结构与产品等级结构没有哪一个是抽象化，哪一个是实现化的分别；相反，它们中的前者是后者的创建者。

因此，抽象工厂模式可以为桥梁模式提供产品创建的结构。在 Java 的 Peer 架构中，就同时使用了抽象工厂模式和桥梁模式，组成了 Java 构件等级结构、Peer 构件等级结构、以及 Toolkit 等级结构这三个部分。

33.7 八戒“转世投胎”的故事

本节从桥梁模式的观点出发，讨论《西游记》中天蓬元帅转世投胎成为猪八戒的故事。



如果存在灵魂的话，应当是抽象化角色，而肉体则是实现化角色。肉体为灵魂的功能提供了实现，就像具体 Java 类为抽象 Java 类提供了实现一样，如右图所示。

八戒是这样描述自己转世投生的过程的：“我本是天河里天蓬元帅……贬下尘凡……投在个母猪胎里，变得这般模样。”八戒投胎，要过六道轮回

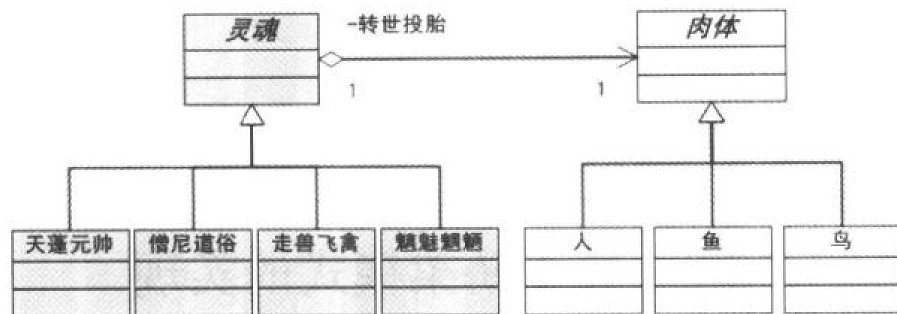


之所："那腾云的身披霞帔……走兽飞禽，魑魅魍魉，滔滔都奔走那轮回之下，各进其道。"

轮回投胎使得芸芸众生的灵魂与肉体之间出现一种弱耦合，使得飞禽走兽可以投胎到与前生不同的肉体中。有了"转世投胎"，才有了天蓬元帅投胎到母猪腹中，投胎为猪的事情。

为什么是桥梁模式

首先，看一看"转世投胎"系统怎样用 UML 语言来表述，如下图所示。



可以看出，"转世投胎"机制将尘世划分成两个等级结构：第一个等级结构是"灵魂"等级结构；第二个等级结构是"肉体"等级结构。前者相当于抽象化，后者相当于实现化。生灵通过功能的委派，调用肉体对象的功能，从而使生灵可以动态地选择自己的肉体。

在上面的 UML 类图中，各个角色与桥梁模式的角色的对应关系如下：

- 抽象化角色：在上面的类图中，"灵魂"是一个抽象类，它给出所有的人、兽、怪、神的灵性的定义，规范出它们的方法和属性。
- 修正抽象化角色：以八戒的例子来说，天蓬元帅的灵魂是八戒的抽象化角色，正如僧尼道俗、走兽飞禽、魑魅魍魉的灵性是它们的抽象化角色一样。这些抽象化角色是具体的抽象化角色，换句话说，它们是在抽象化角色"灵魂"下面的具体化角色，但是它们又是"人"、"鱼"、"鸟"等肉体的抽象化角色。
- 实现化角色：这个角色就是"肉体"。这个抽象类规范具体实现化角色，规定出具体实现化角色应当有的（非私有）方法和属性。
- 具体实现化角色：这就是各种肉体的驱壳，包括人、鱼、鸟、等。

以八戒的情形来讲，没有"转世投胎"，天蓬元帅的前生就和后生没有区别。芸芸众生，飞禽永为飞禽，走兽永为走兽。"转世投胎"代表生灵与肉体的关系，这一点也可以从类图中看到。每个灵性都需要经过滔滔轮回，这意味着抽象化角色"灵魂"持有一个"肉体"的实例，并且构成聚合关系，这个关系就是"转世投胎"。

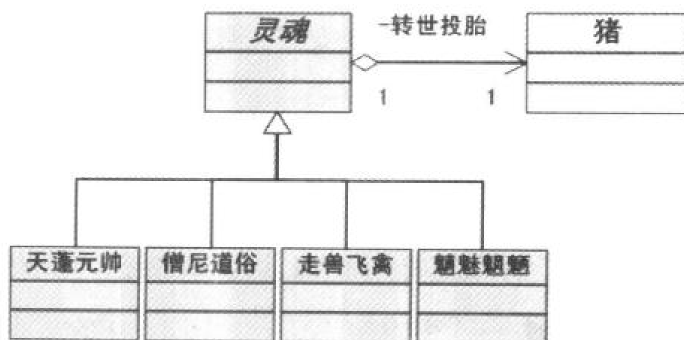
由此可见，"转世投胎"是桥梁模式。

退化的"转世投胎"

如果"肉体"的具体实现只有一种，比如说是猪，那么会怎么样呢？

这样的话，就没有必要设立抽象的肉体角色了，因为反正只有一种"肉体"的实现。

因此，“转世投胎”系统就变成了如下图所示这样的系统。

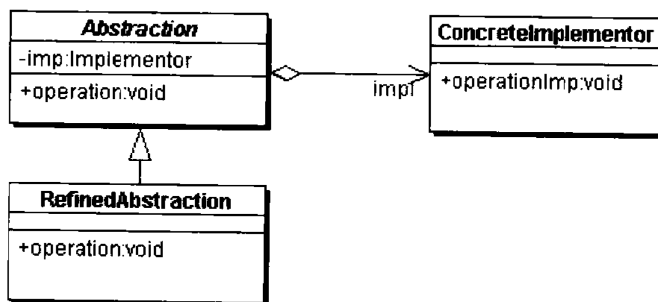


但正如在模式实现的讨论中所说的，把“灵魂”与惟一的“肉体”，即猪分开，仍然是有意义的，因为“灵魂”等级结构可以相对于“肉体”等级结构独立地变化。

33.8 关于桥梁模式的实现

实现化角色的退化

在只有一个具体实现化角色的情况下，抽象的实现化角色就变得没有意义了，不如取消掉，如下图所示。但是抽象化角色与实现化角色的划分仍然有意义。比如，如果实现化角色发生改变，则只需把抽象化角色重新连接到新的实现化角色即可，而不必重新编译客户端。这在一些分散在不同地理位置的系统里，客户端不宜经常改变的情况下是非常有用的。



当具体实现化类只有一个 ConcreteImplementor 的情况下，抽象实现化类就可以省略了。

抽象化角色的行为

在很多情况下，Abstraction 与 RefinedAbstraction 并没有区别。换言之，修正抽象化角色没有“修正”抽象化角色的行为，这当然是允许出现的情况。



Java 的 AWT 库中使用了桥梁角色将 Component 的等级结构与 ComponentPeer 的等级结构分割开，Component 的子类通过委派调用 ComponentPeer 中的功能。在这个例子中，Component 的子类并没有修正 Component 的行为。

多个实现类的情况

当有多于一个实现类时，应该在什么地方、什么时候、怎么创建一个实现类的实例呢？

如果抽象化角色知道具体实现化角色的所有信息，那么它可以在构造子里根据传进的参数来决定创建哪一个具体实现化角色类的实例。

另一个可以采用的方法是，先选用一个具体实现化角色类进行实例化，然后再根据情况改为另一个具体实现层类的实例。比如在读取输入数据时，可以根据使用频率的上升幅度临时决定改用一个对大量数据处理比较合适的实现化角色类。

共享具体实现化角色

可以有几个抽象化角色类合用相同的具体实现化角色类。既然抽象与实现已经分离，那么当然会有几个不同的抽象化共享相同实现的情况。这也是本模式的优越性所在。

在八戒的例子里，天蓬元帅就和野兽共享了具体实现类“猪”。

33.9　在什么情况下应当使用桥梁模式

根据上面的分析，在以下的情况下应当使用桥梁模式：

- 如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。
- 设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。
- 一个构件有多于一个的抽象化角色和实现化角色，系统需要它们之间进行动态耦合。
- 虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

问答题

1. 请问 JDBC/ODBC 桥梁既然叫做“桥梁”，那么它是桥梁模式应用的实例吗？

2. 一个 JavaBean 可以定义为一个可以重复使用的软件构件，它可以在一个可视的编辑环境里修改编辑，所有的 Swing 构件都是 JavaBean。ActiveX 是微软（Microsoft）的架构，与 JavaBean 架构有很多相似之处。

那么，是否可以从 ActiveX 的架构中调用 JavaBean 呢？Sun Microsystem 的 Java 语

言设计师 Graham Hamilton 在 1996 年的 JavaOne 会议上说，这是可以的，只要有一个合适的桥梁作为中介即可。Java 语言很早便给出了这样的桥梁构件，现在它在一个很大的包里免费发送。对这个包感兴趣的读者可以从 http://java.sun.com/products/plugin/index-1.html 下载。

请问 JavaBean 到 ActiveX 的桥梁既然叫做桥梁，那么它是不是桥梁模式的应用呢？

3. 请给出八戒“转世投胎”系统设计图的示意性源代码。

4. 请使用桥梁模式说明 MenuComponent 与 MenuComponentPeer 的关系，并说明在不同的操作系统中，应用系统中的菜单是如何工作的。

5. 一个图像格式有两个基本的方面，一是结构，二是表象。其结构决定图像是怎样存储的，而其表象决定了图像是怎样显示在屏幕上的。对于一个图像格式来说，其结构是在任何一种操作系统中都不变的， 而其表象是在每个操作系统中都有所不同的。比如说，Windows 的 Bitmap 文件在所有的操作系统中都是不变的，而每个操作系统把 Bitmap 文件显示在屏幕上的机制都有所不同。在另一方面，一个操作系统怎样显示一个图像的机制与图像的格式无关。比如，Windows 系统总要把一个图像表示为一个 Bitmap 对象，而不管此图像的源格式。

简单地说，图像的结构和表现是两个不同的方面，应该让它们独立地根据操作系统的不同而不同，桥梁模式正好可以在这里发挥作用。请给出一个示意性的图像浏览器系统，以说明桥梁模式的用法。

问答题答案

1. 桥梁模式的用意是把所考虑的类的抽象化与实现化分开，在这里没有这样的环境。JDBC/ODBC 桥梁不是桥梁模式，而是适配器模式。

由于 JDBC 的接口与 ODBC 的接口不可能相同，因此，JDBC/ODBC 桥梁使用适配器模式改变 ODBC 的 API 为 JDBC 的 API。

关于变压模式，请读者参见本书的“适配器（Adapter）模式”一章。

2. JavaBean 到 ActiveX 的桥梁虽然叫做“桥梁”，但是它不是桥梁模式的应用，而是适配器模式应用的例子。

具体来说，客户端期待 ActiveX 的 API，而 JavaBean 的 API 自然与所期待的 API 不同。从 JavaBean 到 ActiveX 的桥梁便是一个解决方案，它把 JavaBean 的 API 变换成为客户端所期待 ActiveX 的 API。而这符合适配器模式的定义，从而是适配器模式的应用。

3. 所要求的源代码如下所示。

首先是抽象化角色“灵魂”抽象类的源代码，如代码清单 13 所示。

代码清单 13：抽象化角色肉体的示意性源代码

```
abstract public class 灵魂
{
    protected 肉体 imp;
    public void operation()
    {
```



```
        imp.operationImp();
    }
}
```

可以看到，这个抽象类声明并且定义了一个商业方法 operation()，这个方法是通过向实现角色委派实现自己的功能的。

修正抽象类“天蓬元帅”的源代码如代码清单 14 所示。

代码清单 14：修正抽象化角色天蓬元帅的的示意性源代码

```
public class 天蓬元帅 extends 灵魂
{
    // 天蓬元帅的源代码
}
```

抽象实现化角色“肉体”规定出所有的具体实现化角色必须实现的接口。在这里，这个接口就是一个商业方法 operationImp()，如代码清单 15 所示。

代码清单 15：实现化角色肉体的示意性源代码

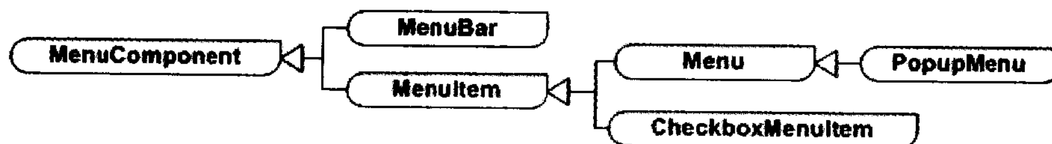
```
abstract public class 肉体
{
    public abstract void operationImp();
}
```

具体实现化角色“猪”实现了抽象实现化角色规定的接口。在这里，这个接口就是商业方法 operation()，如代码清单 16 所示。

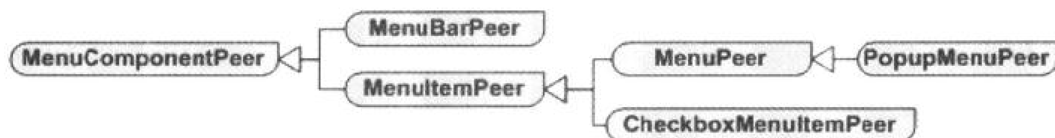
代码清单 16：具体实现化角色猪的示意性源代码

```
public class 猪 extends 肉体
{
    public void operationImp()
    {
        System.out.println("做猪的勾当...");
    }
}
```

4. 与 ComponentPeer 等级结构相独立的，还有一个 MenuComponentPeer 的等级结构。这两个等级结构是完全平行的等级结构的，每个 MenuComponent 对象都有一个 Menu ComponentPeer 类型的对象与之相对应。下图所示为 MenuComponent 等级结构的类图。



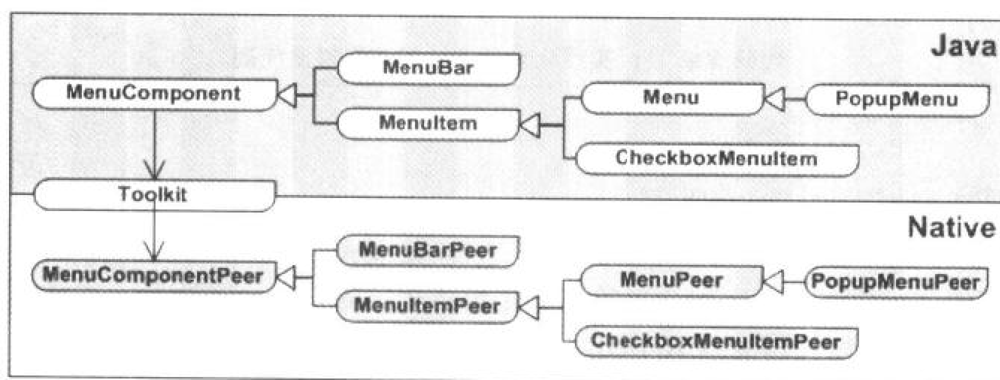
与上面的等级结构完全平行的是 MenuComponentPeer 等级结构。下图所示就是这个结构的类图。



与 Component 以及 ComponentPeer 的情况相同，MenuComponent 扮演了桥梁模式中的抽象化角色，它的子类扮演桥梁模式中的修正抽象化角色；而 MenuComponentPeer 则扮演了桥梁模式中的实现化角色，它的子类扮演了桥梁模式中的具体实现化角色。

在桥梁模式中，每一个修正抽象化角色都对应于一个具体实现化角色。在这里，每一个 MenuComponent 等级结构中的成员都对应于 MenuComponentPeer 等级结构中的成员。

在桥梁模式中，抽象化对象通过向实现化对象的功能委派来实现自己的功能。在这里，每一个 MenuComponent 对象都通过向 MenuComponentPeer 对象的功能委派实现自己的功能，如下图所示。



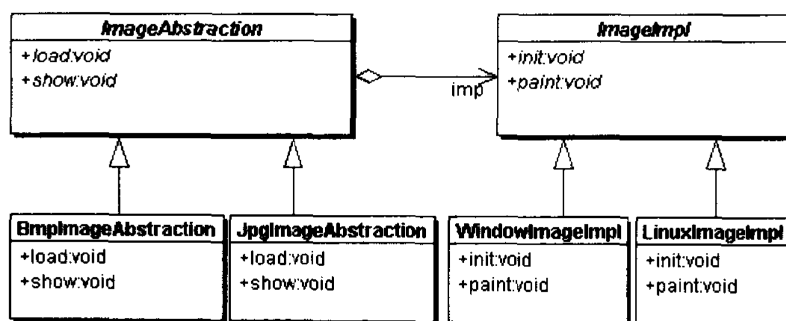
如果程序员是在 WIN 32 系统上开发某个应用菜单系统的，那么程序员看到的菜单是完全 WIN 32 本地风格的菜单；如果是在 UNIX/Motif 系统上运行这个应用程序的话，那么这个用户看到的会是一个 Motif 本地风格的菜单。换言之，应用程序所控制的仅仅是一套抽象的、与操作系统无关的构件，而这套构件的具体实现则是本地的、与操作系统有关的。从抽象构件到本地构件的耦合是动态的弱耦合，或者说，这个耦合不是在编译时期决定的，而是在运行时期动态地决定的。

在应用程序显示一个菜单时，程序所控制的是 MenuComponent 的一个具体子类型，而用户在显示屏上看到的是一个本地的菜单。程序所不知道的，是在 MenuComponent 类型的对象的后面，有一个 MenuComponentPeer 类型的对象。MenuComponent 对象通过向这个本地对象的委派实现自己的功能。

用户针对菜单的任何操作都以事件的方式由操作系统传给某个相应的 MenuComponentPeer 类型的对象，而这个对象将触发对应的 MenuComponent 对象的事件处理机制，从而使应用程序做出反应。

这就是 Java 如何做到既为应用程序提供一个统一的接口，又保持菜单的本地观感的方法。显然，这也是一个桥梁模式的应用。

5. 图像浏览器系统的设计如下图所示。



下面是这个模拟系统的源代码。

首先是作为抽象化角色的抽象类 ImageAbstraction 的源代码，如代码清单 17 所示。

代码清单 17：抽象化角色类的示意性源代码

```
abstract public class ImageAbstraction
{
     protected ImageImpl imp;
   /**
     *  商业方法
     */
     public abstract void load();
   /**
     *  商业方法
     */
     public abstract void show();
}
```

修正抽象化角色类 BmpImageAbstraction 的源代码如代码清单 18 所示。

代码清单 18：抽象化角色类 BmpImageAbstraction 的示意性源代码

```
public class BmpImageAbstraction extends ImageAbstraction
{
   /**
     *  商业方法
     */
     public void load()
     {
          //some code here to load the image into memory
     }
   /**
     *  商业方法
     */
     public void show()
     {
          //some code here to display the image onto screen
     }
}
```

由于 JpgImageAbstraction 类与此相似，故省略。

下面是实现化角色类 ImageImpl 的源代码，如代码清单 19 所示。

代码清单 19：实现化角色类 ImageImpl 的示意性源代码

```
abstract public class ImageImpl
{
    public long width;
    public long height;
    public byte[] data;
    public abstract void init();
    public abstract void paint();
}
```

下面是具体实现化角色 WindowImageImpl 的源代码，如代码清单 20 所示。

代码清单 20：具体实现化角色类 WindowImageImpl 的示意性源代码

```
public class WindowImageImpl extends ImageImpl
{
   /**
    *  商业方法在具体实现化角色的实现
    */
    public void init()
    {
        //load the info of the image
    }
   /**
    *  商业方法在具体实现化角色的实现
    */
    public void paint()
    {
        //display the image onto screen.
    }
}
```

LinuxImageImpl 与此类相似，故略去。

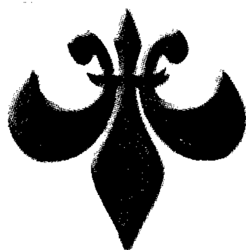


本例子用来解释 Java 语言的跨平台性是怎么实现的，不要使用已有的图像库来实现本例。

参考文献

[LISKOV94] Barbara Liskov and Jeanette Wing. A Behavioral Notion of Subtyping, ACM Transactions on Programming Languages and Systems. Vol 16, No 6, November, 1994, pages 1811-1841

[ZUKOWSKI97] John Zukowski. Java AWT Reference. published by O'Reilly, 1997



第五部分 行为模式

行为模式（Behavioral Pattern）是对在不同的对象之间划分责任和算法的抽象化。行为模式不仅仅是关于类和对象的，而且是关于它们之间的相互作用的。

行为模式分为类的行为模式和对象的行为模式两种。

- 类的行为模式　类的行为模式使用继承关系在几个类之间分配行为。
- 对象的行为模式　对象的行为模式则使用对象的聚合来分配行为。

在后面几章中将要介绍的行为模式包括以下几种：不变模式、策略模式、模版方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、解释器模式、调停者模式等。

此外，与模版方法模式一同介绍的还有“专题：Servlet 技术中的模式”；与观察者模式一同介绍的还有“专题：观察者模式与 AWT 中的事件处理”、“专题：观察者模式与 SAX2 浏览器”、“专题：观察者模式与 Swing 定时器”，以及“专题：MVC 模式与用户输入数据检查”；与迭代子模式一同介绍的还有“专题：Java 对迭代子模式的支持”；与责任链模式一同介绍的还有“专题：定时器和击鼓传花”；与命令模式一同介绍的还有“专题：Swing 库中的命令撤销和恢复”；与状态模式一同介绍的还有“专题：崂山道士与状态模式”；在访问者模式之前介绍的是“专题：单分派和多分派”。

第 34 章　不变（Immutable）模式

一个对象的状态在对象被创建之后就不再变化，这就是所谓的不变模式（Immutable Pattern）。

34.1　引　　言

汉乐府《上邪》说：

上邪！
我欲与君相知，
长命无绝衰。
山无陵，江水为竭，
冬雷震震夏雨雪，
天地合，乃敢与君绝！

显然上面的乐府辞很适合描述不变模式。

一般地讲，一个对象要么是可变对象（Mutable Object），要么是不变对象（Immutable Object）。一个可变对象的状态可以改变，而一个不变对象的状态不可改变。不变模式的做法早在面向对象的编程中便得到使用，是 Mark Grand 在 1998 年出版的[GRAND98]中首次指出这种做法是一个设计模式的，并从设计模式的角度上讨论了不变模式。

不变模式缺少改变自身状态的行为，因此它是关于行为的，所以本书把它划归为行为模式。

34.2　不变模式的结构和实现

不变模式可增强对象的强壮性（robustness）。不变模式允许多个对象共享某一对象，降低了对该对象进行并发访问（Concurrent Access）时的同步化开销。如果需要修改一个不变对象的状态，那么就需要建立一个新的同类型对象，并在创建时将这个新的状态存储在新对象里。

不变模式只涉及到一个类。一个类的内部状态创建后，在整个生命期间都不会发生变化时，这样的类称做不变类。这种使用不变类的做法叫做不变模式。不变模式不需要类图描述。

不变模式有两种形式：一种是弱不变模式，另一种是强不变模式。

弱不变模式

一个类的实例的状态是不可变化的；但是这个类的子类的实例具有可能会变化的状态。这样的类符合弱不变模式的定义。要实现弱不变模式，一个类必须满足下面条件：

第一、所考虑的对象没有任何方法会修改对象的状态；这样一来，当对象的构造子将对象的状态初始化之后，对象的状态便不再改变。

第二、所有的属性都应当是私有的。不要声明任何的公开的属性，以防客户端对象直接修改任何的内部状态。

第三、这个对象所引用到的其它对象如果是可变对象的话，必须设法限制外界对这些可变对象的访问，以防止外界修改这些对象。如果可能，应当尽量在不变对象内部初始化这些被引用到的对象，而不要在客户端初始化，然后再传入到不变对象内部来。如果某个可变对象必须在客户端初始化，然后再传入到不变对象里的话，就应当考虑在不变对象初始化的时候，将这个可变对象复制一份，而不要使用原来的拷贝。

弱不变模式的缺点是：

第一、一个弱不变对象的子对象可以是可变对象；换言之，一个弱不变对象的子对象可能是可变的；这是一个缺点。

第二、这个可变的子对象可能可以修改父对象的状态，从而可能会允许外界修改父对象的状态；这是一个显著的缺点。

文献[GRAND98]所描述的不变模式就是弱不变模式。

强不变模式

一个类的实例的状态不会改变；同时它的子类的实例也具有不可变化的状态。这样的类符合强不变模式。要实现强不变模式，一个类必须首先满足弱不变模式所要求的所有条件，并且还要满足下面条件之一：

第一、所考虑的类所有的方法都应当是 final；这样这个类的子类不能够置换掉此类的方法；

第二、这个类本身就是 final 的，那么这个类就不可能会有子类，从而也就不可能有被子类修改的问题。

文献[BLOCH01]所描述的不变类就是强不变模式。

"不变"和"只读"的区别

"不变"（Immutable）与"只读"（Read Only）是不同的。当一个变量是"只读"时，变量的值不能直接改变，但是可以在其它变量发生改变的时候发生改变。

比如，一个人的出生年月日是"不变"属性，而一个人的年龄便是"只读"属性，但是不是"不变"属性。随着时间的变化，一个人的年龄会随之发生变化，而人的出生年月则不会变化。这就是"不变"和"只读"的区别。



34.3　不变模式在 Java 语言中的应用

不变模式在 Java 语言中有很重要的应用，最著名的应用便是 java.lang.String 类。

String 类

Java 的 String 是一个强不变类，在出现如下的语句时：

```
String youSay = "Object-oriented design means we design, and users object";
String iSay = "Object-oriented design means we design, and users object";
String weSay = "Object-oriented design means we design, and users object";
```

Java 虚拟机其实只会创建这样一个字符串的实例，而这三个 String 对象都在共享这一个值。

如果程序所处理的文字串有频繁的内容变化时，就不宜使用 String 类型，而应当考虑使用 StringBuffer 类型；如果需要对文字串做大量的循环查询时，也不宜使用 String 类型，而应当考虑使用 byte 或 char 数组。

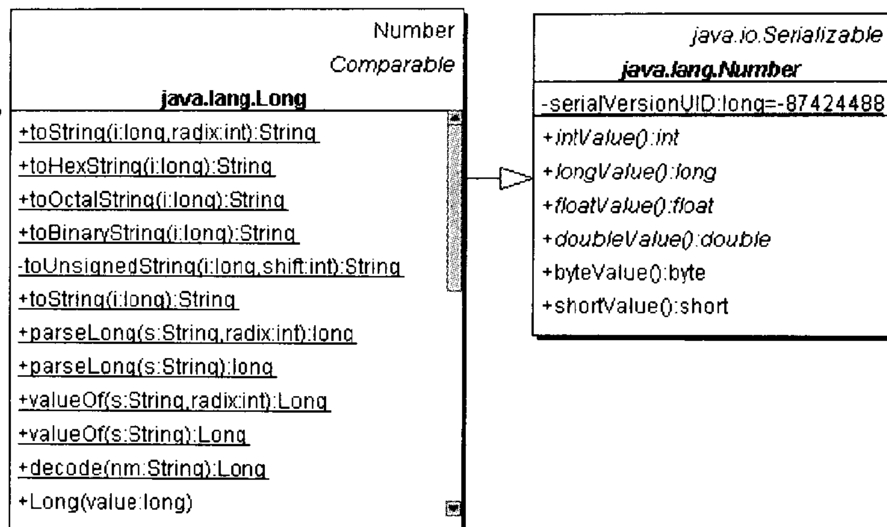
封装类

String 类实际上是一个封装类（Wrapper Class），因为它包装了一个 char 的数组。在 Java 语言中，java.lang 库里还有其他的封装类，如 Integer、Float、Double、Byte、Long、Short、Boolean 和 Character。为什么需要这些封装类呢？

一个 Long 类型的对象所起的作用在于它把一个 long 原始类型的值包装在一个对象里。如果读者使用过 Vector 对象就会知道，存放在 Vector 对象里面的必须是对象，而不可以是原始类型。有了封装类，就可以把原始数据类型包装起来作为对象处理。如果要将一个 long 类型的值存放到一个 Vector 对象里面，就可以把这个 long 类型的值包装到 Long 对象里面，然后再存放到 Vector 对象里。比如：

```
Vector v = new Vector();
v.addElement(new Long(100L));
v.addElement(new Long(101L));
```

这些封装类实际上都是强不变类，因为在这些类都是 final 的，而且在对象被创建时，它们所蕴含的值（也就是它们的状态）就确定了。如果看一看 Long 类型的结构类图就会发现，Long 类型根本没有提供修改内部值的方法，如下图所示。





Long 类的静态方法 parseLong()返回一个新的 Long 对象，而没有修改自己的状态。

34.4 不变模式的优点和缺点

不变模式有很明显的优点：

（1）因为不能修改一个不变对象的状态，所以可以避免由此引起的不必要的程序错误；换言之，一个不变的对象要比可变的对象更加容易维护。

（2）因为没有任何一个线程能够修改不变对象的内部状态，一个不变对象自动就是线程安全的（Thread Safe），这样就可以省掉处理同步化的开销。一个不变对象可以自由地被不同的客户端共享。不变模式惟一的缺点是，一旦需要修改一个不变对象的状态，就只好创建一个新的同类对象。在需要频繁修改不变对象的环境里，会有大量的不变对象作为中间结果被创建出来，再被 Java 语言的垃圾收集器收集走。这是一种资源上的浪费。

在设计任何一个类的时候，应当慎重考虑其状态是否有需要变化的可能性。除非其状态有变化的必要，不然应当将它设计成不变类。

34.5 不变模式与享元模式的关系

享元模式以共享方式支持大量的实例。享元模式中的享元对象可以是不变对象。实际上，大多数的享元对象是不变对象。

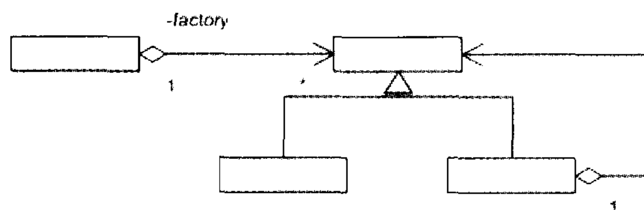
但是，必须指出享元模式并不要求享元对象是不变对象。享元模式要求享元对象的状态不随环境变化而变化，这是使享元对象可以共享的条件。当然如果享元对象成为不变对象的话，是满足享元模式要求的。

享元模式对享元对象的要求是它的内蕴状态与环境无关。这就意味着如果享元对象具



有某个可变的状态，但是只要不会影响享元对象的共享，也是允许的。

不变模式对不变对象的约束较强，而享元模式对享元对象的约束较弱。只要系统允许，可以使用不变模式实现享元对象，但是享元对象不一定非得是不变对象不可。下图所示就是享元模式的简略类图。



34.6　一个例子：复数类

为了说明不变模式的使用，本书在此准备了一个复数类的例子。与前面所讨论过的 Java 语言所提供的封装类，如 Integer 和 Long 类一样，这个复数类是一个不变类。为简化起见，只考虑支持以下的操作：加、减、乘、除；取实部、取虚部、取复数共轭、取相反数；绝对值、相位角、是否相等的测试等。

由于 Java 语言不支持运算符的重载，这些运算必须以静态方法的形式实现。仔细看一看其他的那些方法就会发现，所有的方法都不修改复数值的实部或虚部，如代码清单 1 所示。

代码清单 1：Complex 类的源代码

```
final public class Complex extends Number
        implements java.io.Serializable,  Cloneable, Comparable
{
    /**
     *  虚数单位
     */
    static final public Complex i =  new Complex(0.0, 1.0);
    /**
     *  复数的实部
     */
    private double re;
    /**
     *  复数的虚部
     */
    private double im;
    /**
     *  构造子，根据传进的复数再构造
     *  一个数学值相等的复数
     */
    public Complex(Complex z)
```

```
{
    re = z.re;
    im = z.im;
}
/**
 *  构造子，根据传进的实部和
 *  虚部构造一个复数对象
 */
public Complex(double re, double im)
{
    this.re = re;
    this.im = im;
}
/**
 *  构造子，根据一个实部构造
 *  复数对象
 */
public Complex(double re)
{
    this.re = re;
    this.im = 0.0;
}
/**
 *  默认构造子，构造一个为零
 * 的复数
 */
public Complex()
{
    re = 0.0;
    im = 0.0;
}
/**
 *  把本复数与作为参数传
 *  进的复数相比较
 */
public boolean equals(Complex z)
{
    return (re == z.re   &&   im == z.im);
}
/**
 *  把本对象与作为参数传
 *  进的对象相比较
 */
public boolean equals(Object obj)
{
    if (obj == null)
```



```
    {
        return false;
    }
    else if (obj instanceof Complex)
    {
        return equals((Complex)obj);
    }
    else
    {
        return false;
    }
}
public int hashCode()
{
    long re_bits = Double.doubleToLongBits(re);
    long im_bits = Double.doubleToLongBits(im);
    return (int)((re_bits^im_bits)
        ^((re_bits^im_bits)>>32));
}
/**
 *  返还本复数的实部
 */
public double real()
{
    return re;
}
/**
 *  返还本复数的虚部
 */
public double imag()
{
    return im;
}
/**
 *  静态方法，返还作为参数传
 *  进的复数的实部
 */
public static double real(Complex z)
{
    return z.re;
}
/**
 *  静态方法，返还作为参数传
 *  入的复数的虚部
 */
public static double imag(Complex z)
```

```
{
    return z.im;
}
/**
 *  静态方法，返还作为参数传
 *  入的复数的相反数-z
 */
public static Complex negate(Complex z)
{
    return new Complex(-z.re, -z.im);
}
/**
 *  静态方法，返还作为参数传
 *  入的复数的复共轭
 */
public static Complex conjugate(Complex z)
{
    return new Complex(z.re, -z.im);
}
/**
 *  静态方法，返还两个数的和 x+y.
 */
public static Complex add(Complex x, Complex y)
{
    return new Complex(x.re+y.re, x.im+y.im);
}
/**
 *  静态方法，返还两个数的和 x+y.
 */
public static Complex add(Complex x, double y)
{
    return new Complex(x.re+y, x.im);
}
/**
 *  静态方法，返还两个数的和 x+y.
 */
public static Complex add(double x, Complex y)
{
    return new Complex(x+y.re, y.im);
}
/**
 *  静态方法，返还两个数的差 x-y.
 */
public static Complex subtract(Complex x, Complex y)
{
    return new Complex(x.re-y.re, x.im-y.im);
```

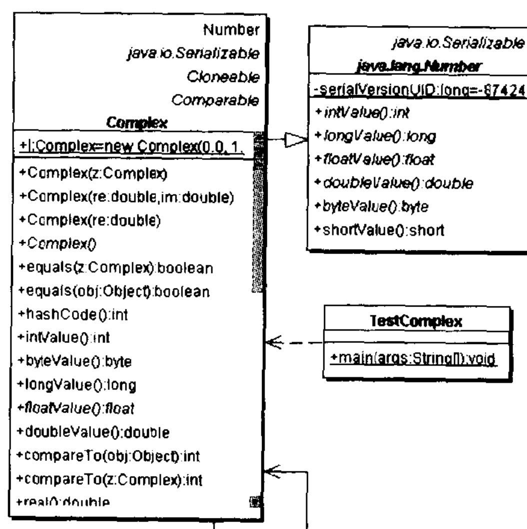


```
}
/**
 *  静态方法，返还两个数的差 x-y.
 */
public static Complex subtract(Complex x, double y)
{
    return new Complex(x.re-y, x.im);
}
/**
 *  静态方法，返还两个数的差 x-y.
 */
public static Complex subtract(double x, Complex y)
{
    return new Complex(x-y.re, -y.im);
}
/**
 *  静态方法，返还两个数的积 x*y.
 */
public static Complex multiply(Complex x, Complex y)
{
    return new Complex(x.re*y.re-x.im*y.im,
        x.re*y.im+x.im*y.re);
}
/**
 *  静态方法，返还两个数的积 x*y.
 */
public static Complex multiply(Complex x, double y)
{
    return new Complex(x.re*y, x.im*y);
}
/**
 *  静态方法，返还两个数的积 x*y.
 */
public static Complex multiply(double x, Complex y)
{
    return new Complex(x*y.re, x*y.im);
}
/**
 *  静态方法，返还两个数的积 x*y.
 */
public static Complex multiplyImag(Complex x, double y)
{
    return new Complex(-x.im*y, x.re*y);
}
/**
 *  静态方法，返还两个数的积 x*y.
```

```
 */
public static Complex multiplyImag(double x, Complex y)
{
    return new Complex(-x*y.im, x*y.re);
}
/**
 *  静态方法，返还两个数的商 x/y.
 */
public static Complex divide(Complex x, Complex y)
{
    double      a = x.re;
    double      b = x.im;
    double      c = y.re;
    double      d = y.im;
    double scale = Math.max(Math.abs(c), Math.abs(d));
    double den = c*c + d*d;
    return new Complex((a*c+b*d)/den, (b*c-a*d)/den);
}
/**
 *  静态方法，返还两个数的商 x/y.
 */
public static Complex divide(Complex x, double y)
{
    return new Complex(x.re/y, x.im/y);
}
/**
 *  静态方法，返还两个数的商 x/y.
 */
public static Complex divide(double x, Complex y)
{
    double      den, t;
    Complex z;
    if (Math.abs(y.re) > Math.abs(y.im))
    {
        t = y.im / y.re;
        den = y.re + y.im*t;
        z = new Complex(x/den, -x*t/den);
    }
    else
    {
        t = y.re / y.im;
        den = y.im + y.re*t;
        z = new Complex(x*t/den, -x/den);
    }
    return z;
}
```

```
    /**
     *  静态方法，返还复数的绝对值 |z|.
     */
    public static double abs(Complex z)
    {
        return z.re * z.re - z.im * z.im;
    }
    /**
     *  静态方法，返还复数的相位角
     */
    public static double argument(Complex z)
    {
        return Math.atan2(z.im, z.re);
    }
    /**
     *  返还复数的字符串
     */
    public String toString()
    {
        if (im == 0.0)
            return String.valueOf(re);
        if (re == 0.0)
            return String.valueOf(im) + "i";
        String sign = ((im < 0.0) ? "" : "+");
        return (String.valueOf(re) + sign + String.valueOf(im) + "i");
    }
    //这里略去了一些方法
}
```

为了节省正文的篇幅，在上面的代码里略去了几个方法，这些方法是抽象类 Number 所要求的。有兴趣的读者可以在后面的问答题答案中找到这些方法的实现。下图所示就是 Complex 类以及抽象类 java.lang.Number 的类图。



显然这个类是 final 类，所以它的所有方法都自动是 final 的。这就是说，这个类不仅是不变类，而且是强不变类。

使用这个 Complex 类的方法很直截了当，下面就是一个例子，演示怎样将两个复数（10+20I）和 i 相乘，如代码清单 2 所示。

代码清单 2：如何使用 Complex 对象

```
public class TestComplex
{
    public static void main(String[] args)
    {
        Complex c1 = new Complex(10,20);
        Complex c2 = new Complex(0, 1);
        Complex res = Complex.multiply(c1,  c2);
        System.out.println("Real part = " + res.real() );
        System.out.println("Imaginary part = " + res.imag() );
    }
}
```

值得注意的是，Complex 对象的相乘和相加不可以使用运算符“+”和“*”，而必须使用 Complex 类中所提供的静态方法。

问答题

1. 中国有一种说法，叫做“专业对口”。请问这是什么模式的应用？
2. 一个 Java 类的构造方法需不需要同步化？
3. 一个子类是否可以置换掉父类的同步方法，它是同步还是不同步呢？
4. 在上面的类里面略去了一些方法，这些方法是实现抽象类 Number 所必需的。请将这些方法补上。
5. 请问 java.awt.Color 类是什么模式？
6. 请问 java.awt.Point 类是否使用了不变模式？如果不是，请改写此类使其遵守不变模式。
7. 请问一个不变类可不可以有带有参数的方法，如代码清单 3 所示的类可不可以成为不变类？

代码清单 3：这个类能否成为一个不变模式

```
public class AClass
{
    public void operation(String param)
    {
        ......
    }
}
```

8. 下面给出一个类的完整源代码，如代码清单 4 所示，请问这个类是不是不变类？

代码清单 4：这是否是一个不变模式的例子

```
public class BClass
{
    private int state = 0 ;

    public int getState()
    {
        return state;
    }

    public void setState(int aState)
    {
        System.out.println("Parameter is "
            + aState);
    }
}
```

问答题答案

1. 所谓的“专业对口”，便是一个人从大学毕业之后，不再改变专业，而应当一辈子在一个专业里工作，这是不变模式的应用。

2. 构造方法不需要同步化，因为它只可能发生在一个线程里，在构造方法返回值前没有其他线程可以使用该对象。

3. 一个子类可以置换掉父类的同步方法，使它同步或不同步。这就是说，子类的方法不继承其父类的方法的特性。父类的方法不改变，如果明显地调用父类的同步方法，那么这个方法将是同步调用的。

4. 这些方法的源代码如代码清单 5 所示。

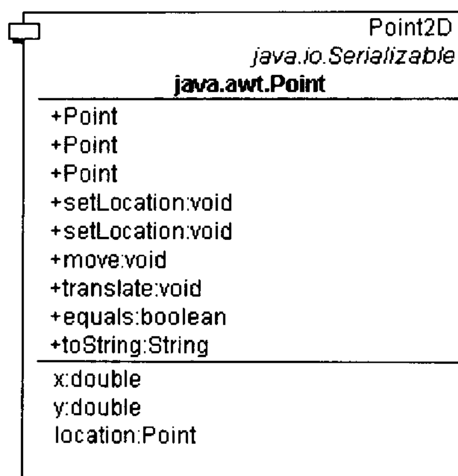
代码清单 5：Complex 对象需要提供抽象类 Number 的几个方法

```
...
/**
 *     将复数值的实部以 int 返还
 */
public int intValue()
{
    return (int)re;
}
/**
 *     将复数值的实部以 byte 返还
 */
public byte byteValue()
{
    return (byte)re;
}
```

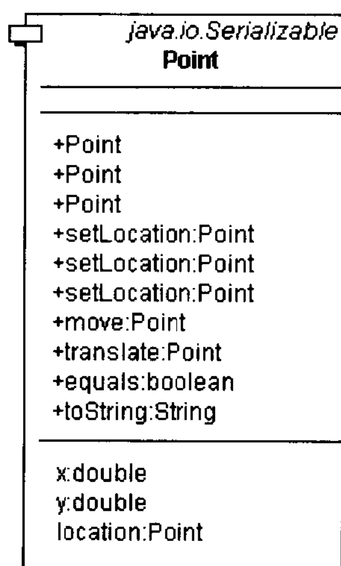
```
/**
 *    将复数值的实部以 long 返还
 */
public long longValue()
{
    return (long)re;
}
/**
 *    将复数值的实部以 float 返还
 */
public float floatValue()
{
    return (float)re;
}
/**
 *    将复数值的实部以 double 返还
 */
public double doubleValue()
{
    return re;
}
/**
 *    将本复数与另一个对象比较
 */
public int compareTo(Object obj)
{
    return compareTo((Complex)obj);
}
/**
 *    比较两个复数对象
 */
public int compareTo(Complex z)
{
    int     compare = new Double(re)
        .compareTo(new Double(z.re));
    if (compare == 0)
    {
        compare = new Double(im)
            .compareTo(new Double(z.im));
    }
    return compare;
}
...
```

5. java.awt.Color 类的每一个方法都是改变对象的内部状态的，因此它使用的是不变模式。

6. java.awt.Point 不是不变模式，因为 Point 类提供了几个改变对象状态的方法，比如 translate(int x, int y)方法将点的位置移动 *x* 和 *y* 个单位，其类图如下图所示。



一个遵守不变模式的 Point 类必须改写所有的改值方法，让它们返还一个新的 Point 对象。这个新的 Point 类如下图所示。



这个新的 Point 类的源代码如代码清单 6 所示。

代码清单 6：遵守不变模式的 Point 类

```
public class Point implements
    java.io.Serializable
{
    public int x;
    public int y;
    public Point()
    {
        this(0, 0);
```

```
}
public Point(Point p)
{
    this(p.x, p.y);
}
/**
 * Returns the X coordinate of
 * the point in double precision.
 */
public double getX()
{
    return x;
}
/**
 * Returns the Y coordinate of
 * the point in double precision.
 */
public double getY()
{
    return y;
}
/**
 * 返还点对象
 */
public Point getLocation()
{
    return new Point(x, y);
}
/**
 * 将点移到(x,y)
 */
public Point setLocation(Point p)
{
    return new Point(p.x, p.y);
}
/**
 * 将点移到(x,y)
 */
public Point setLocation(int x, int y)
{
    return new Point(x, y);
}
/**
 * 将点移到(x,y)
 */
public Point setLocation(double x, double y)
```

```
    {
        return new Point((int) Math.round(x),
            (int) Math.round(y));
    }
    /**
     * 将点移到(x,y)
     */
    public Point move(int x, int y)
    {
        return new Ponit(x, y);
    }
    /**
     * 将点移动 x 和 y 单位
     */
    public Point translate(int x, int y)
    {
        return new Point(this.x + x, this.y + y);
    }
    public boolean equals(Object obj)
    {
        if (obj instanceof Point)
        {
            Point pt = (Point)obj;
            return (x == pt.x) && (y == pt.y);
        }
        return super.equals(obj);
    }
    public String toString()
    {
        return getClass().getName()
            + "[x=" + x + ",y=" + y + "]";
    }
}
```

由于上面的这个类所给出的方法可以被其子类置换掉，因此这个类是弱不变类。当然，要想将它改为强不变类并不困难，只需要将这个类声明为 final 就可以了。

7. 由于没有提供完整的源代码，因此看不出这个类现在是不是不变类，但是这个类可以成为不变类。

一个不变对象的内部状态不可以改变，但这并不意味着不可以有外部参量传入到一个方法中，只要这个外部参量不改变对象的内部状态就可以。

下面的实现就是一个真正的（弱）不变类，如代码清单 7 所示。

代码清单 7：一个不变模式的例子

```
public class AClass
{
    public void operation(String param)
```

```
        {
            System.out.println("Incoming parameter = " + param);
        }
    }
```

8. 这个类确实是一个（弱）不变类。这个类有一个状态，存储在 Int 原始类型的 State 属性里面。粗看上去它有一个改值方法 setState()，但是这个方法实际上并没有修改这个状态的值，因此这个类确实是一个不变类。

由于 setState()并不是一个真正的改值方法，建议将这个方法改名，以免发生不必要的混淆。

参考文献

[BLOCH01] Joshua Bloch. Effective Java – Programming Language Guide. published by Addison-Wesley，2001

第 35 章　策略（Strategy）模式

策略模式属于对象的行为模式[GOF95]。其用意是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。

35.1　引　　言

问题

假设现在要设计一个贩卖各类书籍的电子商务网站的购物车（Shopping Cart）系统。一个最简单的情况就是把所有货品的单价乘上数量，但是实际情况肯定比这要复杂。比如，本网站可能对所有的教材类图书实行每本一元的折扣；对连环画类图书提供每本 7%的促销折扣，而对非教材类的计算机图书有 3%的折扣；对其余的图书没有折扣。由于有这样复杂的折扣算法，使得价格计算问题需要系统地解决。

那么怎么解决这个问题呢？

解决方案

这里提供三种方案如下所示：

（1）所有的业务逻辑都放在客户端里面。客户端利用条件选择语句决定使用哪一个算法。这样一来，客户端代码会变得复杂和难以维护。

（2）客户端可以利用继承的办法在子类里面实现不同的行为，但是这样会使得环境和行为紧密耦合在一起。强耦合会使两者不能单独演化。

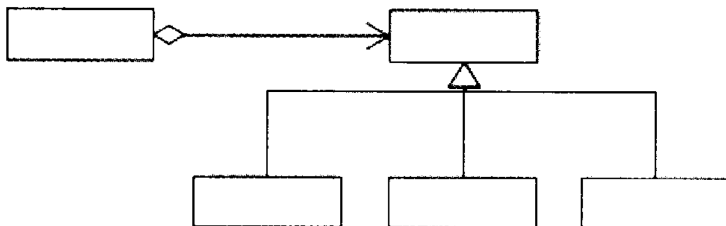
（3）使用策略模式。策略模式把行为和环境分割开来。环境类负责维持和查询行为类，各种算法则在具体策略类（ConcreteStrategy）中提供。由于算法和环境独立开来，算法的增减、修改都不会影响环境和客户端。

策略模式正是解决这个问题的系统化方法。当出现新的促销折扣或现有的折扣政策出现变化时，只需要实现新的策略类，并在客户端登记即可。策略模式相当于“可插入式（Pluggable）的算法”。

当准备在一个系统里使用策略模式时，首先必须找到需要包装的算法，看看算法是否可以从环境中分割开来，最后再考察这些算法是否会在以后发生变化。

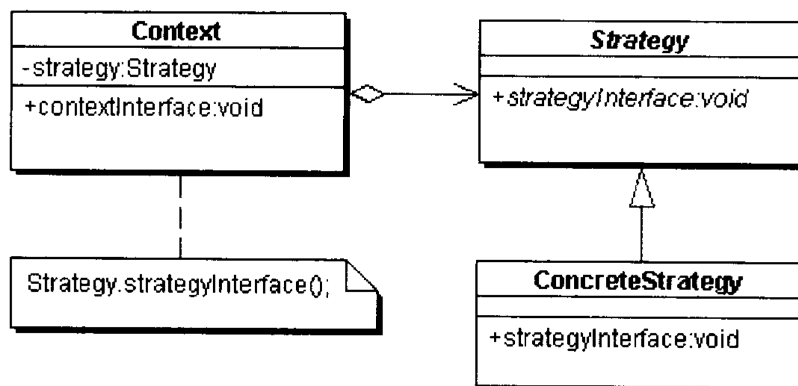
引进策略模式

策略模式是对算法的包装，是把使用算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是："准备一组算法，并将每一个算法封装起来，使得它们可以互换。"策略模式的简略类图如下图所示。



策略模式的结构

策略又称做政策（Policy）模式[GOF95]。下面就以一个示意性的实现讲解策略模式实例的结构，如下图所示。



这个模式涉及到三个角色：

- 环境（Context）角色：持有一个 Strategy 类的引用。
- 抽象策略（Strategy）角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
- 具体策略（ConcreteStrategy）角色：包装了相关的算法或行为。

首先看一看环境角色的源代码，如代码清单 1 所示，这个类持有一个对策略角色的引用。

代码清单 1：环境类的源代码

```
public class Context
{
    private Strategy strategy;
```



```
    /**
    *  策略方法
    */
    public void contextInterface()
    {
        strategy.strategyInterface();
    }
}
```

抽象策略类的源代码如代码清单 2 所示，这个抽象类可以用 Java 接口取代。抽象策略类规定所有具体策略类必须实现的接口。

代码清单 2：抽象策略类的源代码

```
abstract public class Strategy
{
    /**
    *  策略方法
    */
    public abstract void strategyInterface();
}
```

具体策略类实现了抽象策略类所声明的接口，如代码清单 3 所示。

代码清单 3：具体策略类的源代码

```
public class ConcreteStrategy extends Strategy
{
    /**
    *  策略方法
    */
    public void strategyInterface()
    {
        //write you algorithm code here
    }
}
```

这里所给出的仅仅是策略模式的最小实现，因此具体策略角色才只有一个。一般而言，有意义的策略模式的应用都会涉及到多于一个的具体策略角色。读者可以把这里给出的源代码当做策略模式的骨架，在将它使用到自己的系统中时，还需要添加上与系统有关的逻辑。

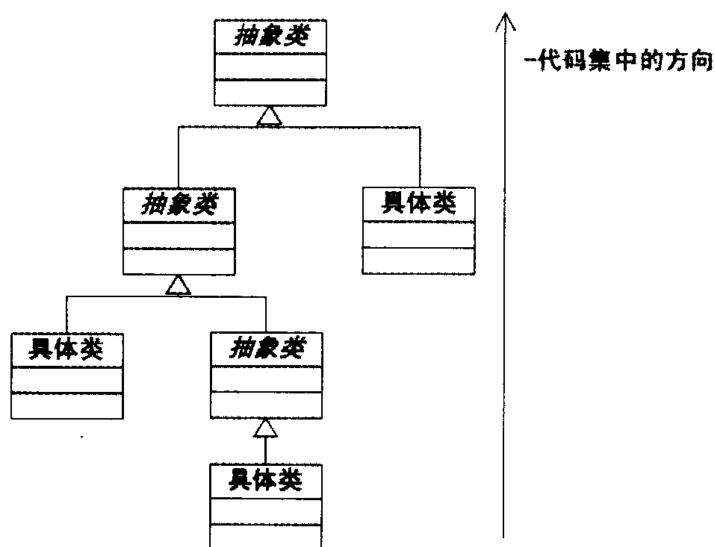
35.2　模式的实现

策略模式的实现有以下这些需要注意的地方。

（1）经常见到的是，所有的具体策略类都有一些公有的行为。这时候，就应当把这

些公有的行为放到共同的抽象策略角色 Strategy 类里面。当然这时候抽象策略角色必须要用 Java 抽象类实现，而不能使用 Java 接口。

这其实也是典型的将代码向继承等级结构的上方集中的标准做法。代码集中的方向如下图所示，并请阅读本书的“抽象类”一章。



（2）策略模式在每一个时刻都只能使用一个策略对象，但是有的时候一个应用程序同时与几个策略对象相联系。换言之，在应用程序启动时，所有的策略对象就已经被创立出来，而应用程序可以在几个策略对象之间调换。

这只有在策略对象不会耗费很多计算机内存资源的情况下才可行，只有在策略对象的初始化会花费很长时间的情况下才需要。

35.3 Java 语言内部的例子

Java 语言使用了很多的设计模式，包括策略模式。使用策略模式的例子可以在 java.awt 库和 Swing 库中看到。

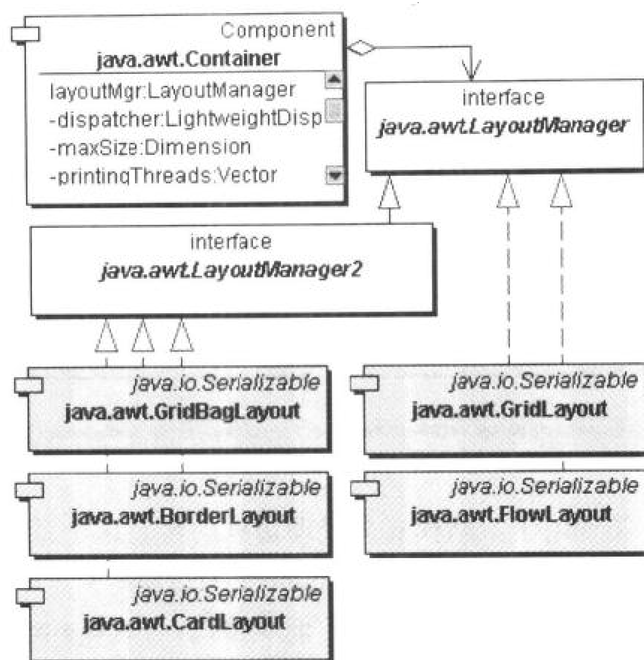
AWT 中的 LayoutManager

java.awt 类库需要在运行期间动态地由客户端决定一个 Container 对象怎样排列它所有的 GUI 构件。Java 语言提供了几种不同的排列方式，包装在不同的类里：

- BorderLayout
- FlowLayout
- GridLayout
- GridBagLayout
- CardLayout



LayoutManager 的类图如下图所示。



图中有阴影的类是具体策略角色，java.awt.Container 类是环境角色，而 java.awt.LayoutManager 则是抽象策略角色。

任何人都可以设计实现自己的 Layout 类。比如 Borland 公司就在 JBuilder 里面提供了 XYLayout，作为对几种 JDK 提供的 Layout 类的补充。

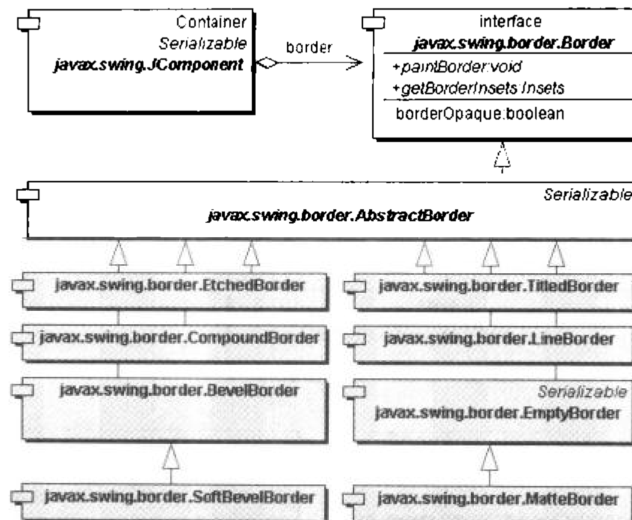
这个系统涉及到三个角色：

- 环境（Context）角色：在这里由 Container 类扮演。
- 抽象策略（Strategy）角色：这是一个抽象角色，由 LayoutManager 类扮演。此角色给出所有的具体 Layout 类所需的接口。
- 具体策略（ConcreteStrategy）角色：由 BorderLayout、FlowLayout、GridLayout、GridBagLayout、CardLayout 等扮演（请见图中有阴影的类），它们包装了不同的 Layout 行为。

Swing 中的 Border

在任何一个 Swing 构件上都可以画上边框（Border），比如 panel、button 等。而 Swing 库提供了很多的边框类型，包括 bevel、line、titled 以及 CompoundBorder 类等。Swing 构件的基类是 JComponent 类，而这个类负责为 Swing 构件画上边框。

JComponent 类实现了 paintBorder()方法，并且保持一个私有的对边框对象的引用。由于 Border 是一个接口而不是具体类，因此，这个引用可以指向任何实现了 Border 接口的边框对象。Swing 中各种 Border 的类图如下图所示。



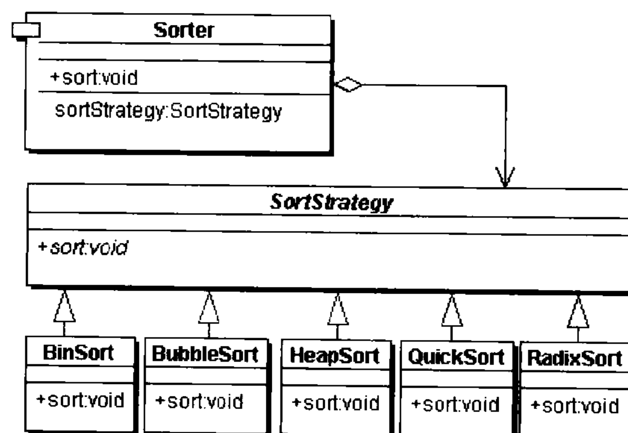
显然，JComponent 类使用了策略模式来处理边框问题。这个系统涉及到三个角色：

- 环境（Context）角色：在这里由 JContainer 类扮演。
- 抽象策略（Strategy）角色：这是一个抽象角色，由 Border 类扮演。此角色给出所有的具体边框类所需的接口。
- 具体策略（ConcreteStrategy）角色：由 BevelBorder、SoftBevelBorder、CompoundBorder、EtchedBorder、TitledBorder、LineBorder、EmptyBorder 以及 MatteBorder 等扮演（请见图中有阴影的类），它们包装了不同的边框行为。

35.4 排序策略系统

假设要设计一个排序系统（Sorter System），动态地决定采用二元排序（Binary Sort）、泡沫排序（Bubble Sort）、堆栈排序（Heap Sort）、快速排序（Quick Sort）、基数排序（Radix Sort）。

显然，采用策略模式把几种排序算法包装到不同的算法类里面，让所有的算法类具有相同的接口，就是一个很好的设计。排序策略系统的设计如下图所示。



客户端必须决定在何时使用哪一个排序算法，换言之，这个决定不是在模式内部决定的。

35.5　一个例子：图书折扣的计算

图书折扣的算法

本节回到本章开头所提出的图书销售的折扣计算问题。根据描述，折扣是根据以下的几个算法中的一个进行的：

算法一：对有些图书没有折扣。折扣算法对象返还 0 作为折扣值。

算法二：对有些图书提供一个固定量值为 1 元的折扣。

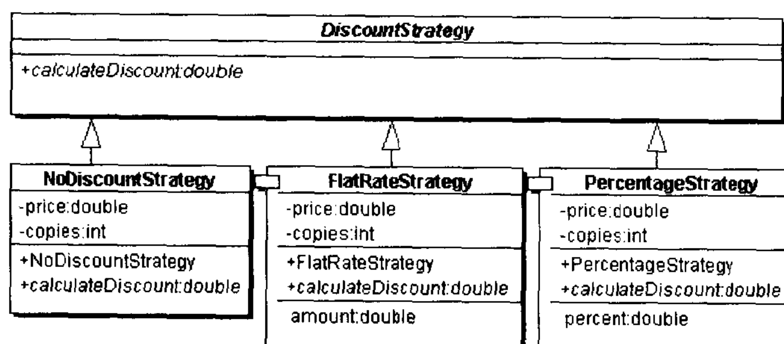
算法三：对有些图书提供一个百分比的折扣，比如一本书价格为 20 元，折扣百分比为 7%，那么折扣值就是 20×7%=1.4（元）。

使用策略模式描述的话，这些不同的算法都是不同的具体策略角色：用一个 NoDiscountStrategy 对象描述“算法一”；用一个 FlatRateStrategy 对象描述“算法二”；用一个 PercentageStrategy 对象描述“算法三”。

为了定义出算法类型的等级结构，需要一个抽象策略角色作为等级结构的超类型。这个抽象策略角色由一个 Java 抽象类 DiscountStrategy 扮演。

系统类图

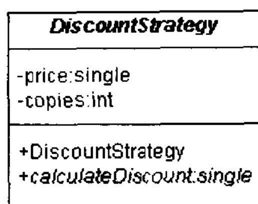
图书折扣系统的设计图如下图所示。



下面就仔细看一看各个角色的结构。

抽象折扣类

可以看出，DiscountStrategy 是一个抽象类，它定义出一个类型等级结构（Type Hierarchy）。抽象折扣类的结构图如下图所示。



抽象折扣类的源代码如代码清单 4 所示。

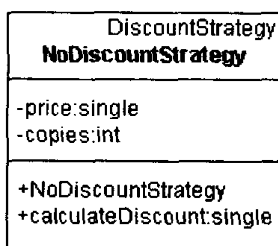
代码清单 4：抽象折扣角色的源代码

```
package com.javapatterns.strategy.booksales;
abstract public class DiscountStrategy
{
    private single price = 0;
    private int copies = 0;
   /**
    *  策略方法
    */
     public abstract single calculateDiscount();
    /**
    *  构造子
    */
    public DiscountStrategy(single price, int copies)
    {
        this.price = price;
        this.copies = copies;
    }
}
```

当然由于本类仅仅是一个抽象类，而一个抽象类不可能有实例，因此本类给出的接口是供具体子类实现的。可以看出，在使用具体折扣类时，客户端应当调用构造子并且传入图书的单价、购买的册书，然后调用 calculateDiscount()方法得到折扣总额。

具体折扣类 NoDiscountStrategy

具体折扣类 NoDiscountStrategy 是折扣类型等级结构中的成员。作为抽象类 DiscountStrategy 的子类，它实现了 calculateDiscount()方法。具体折扣类 NoDiscountStrategy 的结构图如下图所示。



具体折扣子类 NoDiscountStrategy 的源代码如代码清单 5 所示。

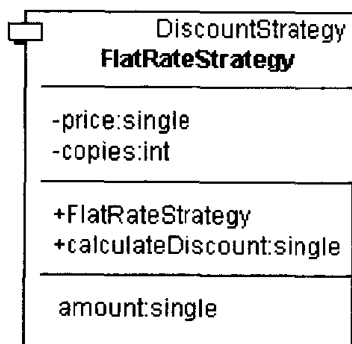
代码清单 5：具体折扣子类 NoDiscountStrategy

```
package com.javapatterns.strategy.booksales;
public class NoDiscountStrategy extends DiscountStrategy
{
    private single price = 0;
    private int copies = 0;
    /**
    *  构造子
    */
    public NoDiscountStrategy(single price, int copies)
    {
        this.price = price;
        this.copies = copies;
    }
   /**
    *  策略方法
    */
     public single calculateDiscount()
    {
        return 0;
    }
}
```

实际上对任何输入都给出 0 作为折扣总值，这表明对该类图书没有折扣。

具体折扣类 FlatRateStrategy

具体折扣类 FlatRateStrategy 如下图所示。



此类的源代码如代码清单 6 所示。

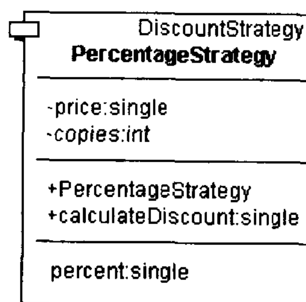
代码清单 6、具体折扣类 FlatRateStrategy 的源代码

```
package com.javapatterns.strategy.booksales;
public class FlatRateStrategy extends DiscountStrategy
{
```

```
    private single price = 0;
    private int copies = 0;
    private single amount;
    /**
    *  构造子
    */
    public FlatRateStrategy(single price, int copies)
    {
        this.price = price;
        this.copies = copies;
    }
    public single getAmount()
    {
        return amount;
    }
    public void setAmount(single amount)
    {
        this.amount = amount;
    }
    /**
    *  策略方法
    */
    public single calculateDiscount()
    {
        return copies * amount;
    }
}
```

具体折扣类 PercentageStrategy

具体折扣类 PercentageStrategy 的结构图如下图所示。



此类的源代码如代码清单 7 所示。

代码清单 7：具体折扣类 PercentageStrategy 的源代码

```
package com.javapatterns.strategy.booksales;
```

```
public class PercentageStrategy extends DiscountStrategy
{
    private single percent;
    private single price = 0;
    private int copies = 0;
    /**
    *   构造子
    */
    public PercentageStrategy(single price, int copies)
    {
        this.price = price;
        this.copies = copies;
    }
    public single getPercent()
    {
        return percent;
    }
    public void setPercent(single percent)
    {
        this.percent = percent;
    }
   /**
    *   策略方法
    */
     public single calculateDiscount()
    {
        return copies * price * precent;
    }
}
```

何时使用何种具体策略角色

在学习策略模式时，学员常问的一个问题是：为什么不能从策略模式中看出哪一个具体策略适用于哪一种情况呢？

答案非常简单，策略模式并不负责做这个决定。换言之，应当由客户端自己决定在什么情况下使用什么具体策略角色。以本节所讨论的系统为例，什么图书应当使用什么折扣方案并不是策略模式所能解决的，这个决定应当由客户端决定。

策略模式仅仅封装算法，提供新算法插入到已有系统中，以及老算法从系统中“退休”的方便，策略模式并不决定在何时使用何种算法。

同时嵌套使用多于一个的算法

策略模式不适合于处理同时嵌套多于一个算法的情形。

一般而言，策略模式只适用于客户端在几种算法中选择一种的情形，并不适用于客户

端同时需要几种算法的情形。比如，如果在本节给出的例子中假设一个新的法则：

算法四：在所有的折扣算法计算后，总的折扣总不能超过 1000 元。

这就意味着客户端必须首先使用折扣算法一、二、三计算出折扣总值后，再使用算法四。这种重复使用多种算法的情形不是单纯的策略模式可以处理的，需要进一步使用装饰模式。本书建议读者阅读“装饰模式”一章。

35.6 在什么情况下应当使用策略模式

在下面的情况下应当考虑使用策略模式：

（1）如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。

（2）一个系统需要动态地在几种算法中选择一种。那么这些算法可以包装到一个个的具体算法类里面，而这些具体算法类都是一个抽象算法类的子类。换言之，这些具体算法类均有统一的接口，由于多态性原则，客户端可以选择使用任何一个具体算法类，并只持有一个数据类型是抽象算法类的对象。

（3）一个系统的算法使用的数据不可以让客户端知道。策略模式可以避免让客户端涉及到不必要接触到的复杂的和只与算法有关的数据。

（4）如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。此时，使用策略模式，把这些行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句，并体现面向对象设计的概念。

35.7 策略模式的优点和缺点

策略模式有很多优点和缺点。它的优点有：

（1）策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免重复的代码。

（2）策略模式提供了可以替换继承关系的办法。继承可以处理多种算法或行为。如果不是用策略模式，那么使用算法或行为的环境类就可能会有一些子类，每一个子类提供一个不同的算法或行为。但是，这样一来算法或行为的使用者就和算法或行为本身混在一起。决定使用哪一种算法或采取哪一种行为的逻辑就和算法或行为的逻辑混合在一起，从而不可能再独立演化。继承使得动态改变算法或行为变得不可能。

（3）使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重转移语句里面，比使用继承的办法还要原始和落后。

策略模式的缺点有：

（1）客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于

客户端知道所有的算法或行为的情况。

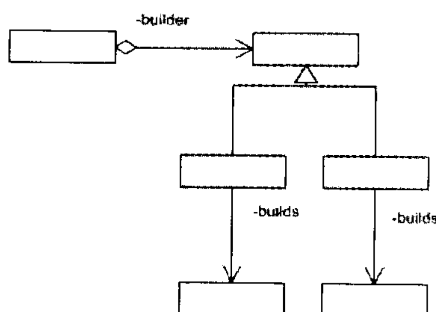
（2）策略模式造成很多的策略类。有时候可以通过把依赖于环境的状态保存到客户端里面，而将策略类设计成可共享的，这样策略类实例可以被不同客户端使用。换言之，可以使用享元模式来减少对象的数量。

35.8　策略模式与其他模式的关系

和策略模式有关系的其他模式有适配器模式、享元模式和模版方法模式。

策略模式与建造（Builder）模式的关系

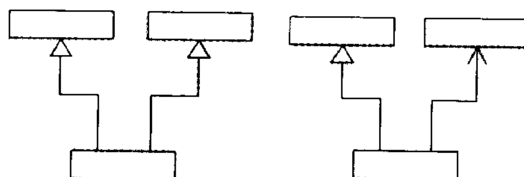
策略模式在结构上与建造模式相似。事实上，可以把建造模式看做策略模式的一个特殊情况。建造模式的类图结构如下图所示。



存在于这两种模式之间的区别是它们在什么时候使用、在什么情况下使用，以及模式的功能。在建造模式里面，核心的功能是以一步一步的方式创立一个产品，它的 Director 角色一遍又一遍地调用建造者对象来把零件增加到产品上，最后返还整个产品对象。而在策略模式里面，Strategy 角色在 Context 角色的调用下，不仅仅可以提供新对象的创立，而且可以提供任何一种运行状态的服务。

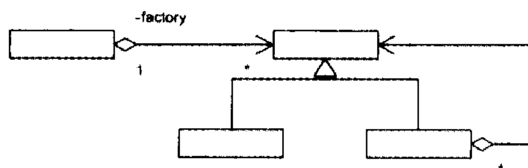
策略模式与适配器（Adapter）模式的关系

适配器模式在结构上与策略模式相似，它们之间的区别在于它们的用意不同。适配器模式的用意是允许一个客户对象通过调用一个配备着完全不同的接口的对象来完成它原来所要做的功能。策略模式的用意是使用统一的接口为客户端提供不同的算法。适配器模式的简略类图如下图所示，左边的是类的适配器模式，右边的是对象的适配器模式。



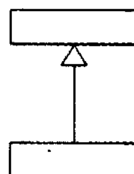
策略模式与享元模式（Flyweight Pattern）的关系

如果有多个客户端对象需要调用同样的一些策略类的话，就可以使它们实现享元模式，这样客户端可以共享这些策略类。享元模式的简略类图如下图所示。



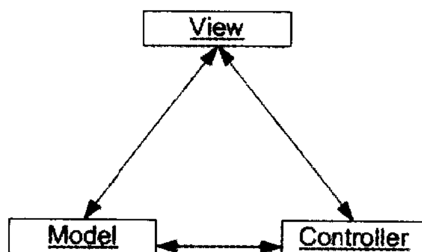
策略模式与模版方法（Template）模式的关系

模版方法模式与策略模式的不同在于，策略模式使用委派的方法提供不同的算法行为，而模版方法模式使用继承的方法提供不同的算法行为。模版方法模式的结构如右图所示。



策略模式与 MVC 模式的关系

读者可以从“MVC 模式”一章看到关于 MVC 模式的专门讨论，这里仅就策略模式与 MVC 模式的关系进行一个简短的讨论。MVC 模式的结构如下图所示。



在 MVC 模式里面有三个角色：模型（Model），视图（View）和控制器（Controller）。正如观察者模式是 MVC 模式的具体实例一样，策略模式经常被用到 MVC 模式里面。

在 MVC 模式里的视图–控制器关系就是典型的策略模式的例子。视图常常是一个可视构件，而控制器需要对用户界面的事件做出反应。如当用户单击一个按键构建后，控制器需要首先使按键处于禁止（Disabled）状态，然后运行按键所代表的操作，再重新允许按键事件发生。

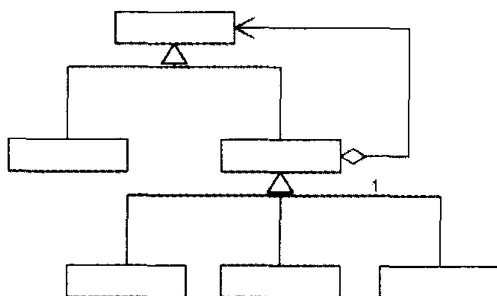
这里按键所代表的操作可以根据视窗上用户所做选择的改变而改变，这就是策略模式的应用。

策略模式与装饰（Decorator）模式的关系

策略模式和装饰模式是不同的，装饰模式的用意是在不改变接口的情况下，增强一个

对象的功能；而策略模式在保持接口不变的情况下，使具体算法可以互换。

策略模式只能处理客户端从几个具体算法中选择一个的情形，而不能处理客户端同时选用一种以上的算法的情形。为了处理后者，可以同时使用策略模式和装饰模式。装饰模式的静态图如下图所示。



35.9　设计原则的讨论

“开–闭”原则

正如本书在“开–闭原则”一章中描述的，策略模式是“开–闭”原则的一个极好的应用范例。

在采用策略模式之前，设计师必须从“开–闭原则”出发，考察这个图书销售系统是否有可能在将来引入新的折扣算法。如果有可能，那么就应当将所有的折扣算法封装起来，因为它们是可变化的因素。系统必须能够在新的算法出现时，很方便地将新的算法插入到已有的系统中，而不必修改已有的系统。

关于“开–闭”原则的详细讨论，请读者阅读本书的“开–闭原则”一章。

里氏代换原则

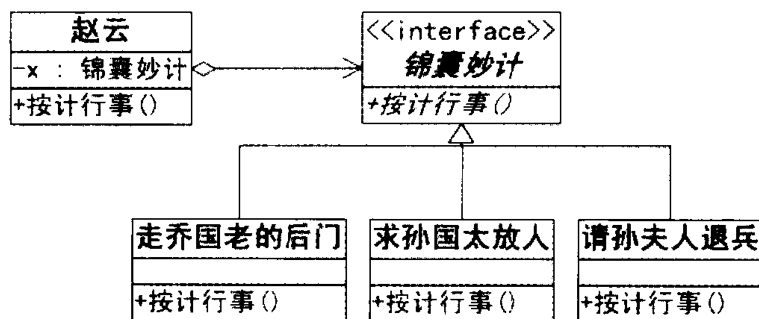
正如本书在“里氏代换原则”一章中描述的，策略模式之所以可行的基础便是里氏代换原则。

策略模式要求所有的策略对象都是可以互换的，因此它们都必须是一个抽象策略角色的子类。在客户端则仅知道抽象策略角色类型，虽然变量的真实类型可以是任何一个具体策略角色的实例。

35.10　诸葛亮的锦囊妙计

最后，本书拟以策略模式的角度出发重新探讨《三国演义》中的一段故事。

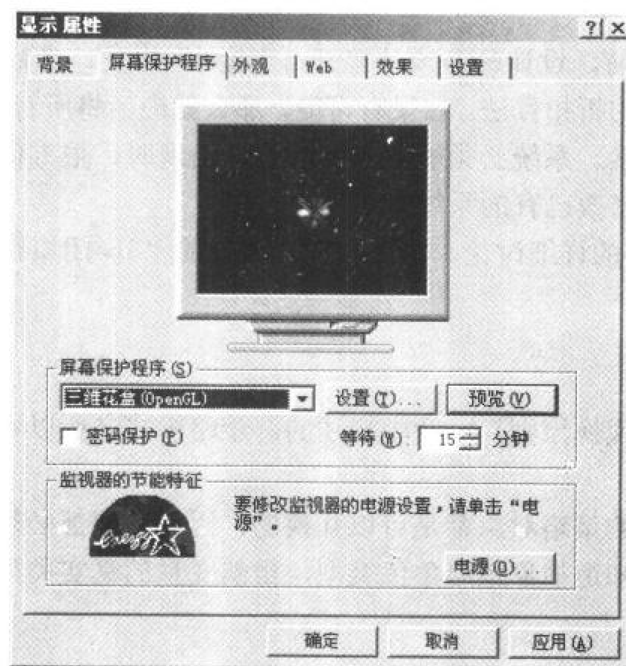
当年赵云保护刘备入吴国迎娶美女，诸葛亮给了他三个锦囊，囊中有三条妙计，嘱咐赵云伺机行事。这三个锦囊妙计很符合策略模式的定义。诸葛亮即便不是策略模式的始作俑者，也应当是策略模式的早期实践者。诸葛亮的三个锦囊妙计的静态设计图如下图所示。



可以看出，赵云扮演了环境角色，而“走乔国老的后门”、“求孙国太放人”和“请孙夫人退兵”则是具体策略角色。诸葛亮设定这些“锦囊妙计”的做法便是抽象策略角色，它规定了具体的策略类的接口。

Windows 屏幕保护程序的设置

每一个屏幕保护程序都有不同的参数需要设定，如下图所示，此“显示属性”窗口选择“三维花盒”作为屏幕保护程序。



每一种程序提供不同的显示算法，在屏幕上放映出文字、二维或三维的图形。用户必须可以为每一种程序设定所需要的参数。策略模式提供了解决这个问题的方案。当用户选中一个屏幕保护程序后，就需要设定程序的参数。“三维花盒”程序需要下图所示的参数。





这是典型的策略模式的应用：每一个屏幕保护程序都有一个自己的参数设定窗口。这个窗口便是具体策略角色，它代表不同的算法；用户可以选择一个算法并设定其参数；所有的参数设定窗口都有非常自恰的界面设计，让用户相信这些独立的策略对象后面有一个共同的抽象接口约束它们，它就是抽象策略角色。

类似的做法还可以在打印机的参数设定里找到。

问答题

1. 请给出排序算法系统的示意性源代码。

2. 诸葛亮在临死前授予杨仪一个锦囊，密嘱曰："我死，魏延必反，……那时自有斩魏延之人也。"同时授马岱以密计，只待魏延喊叫时，便出其不意斩之。……杨仪读罢锦囊计策，已知伏下马岱在彼，故依计而行，果然杀了魏延。后人有诗曰："诸葛先机识魏延，已知日后反西川。锦囊遗计人难料，却见成功在马前。"

请问这是策略模式吗？如果是，请给出类图定义；如果不是，请说明为什么不是，以及什么模式更接近这个故事。

问答题答案

1. 这里给出系统的所有示意性源代码，如代码清单 8 所示。Sorter 类扮演了策略模式中的环境角色。

代码清单 8：环境角色 Sorter 的源代码

```
public class Sorter
{
    private SortStrategy sortStrategy;

   /**
    *  策略方法
    */
    public void sort()
    {
        sortStrategy.sort();
```

```
    }
    public void setSortStrategy(SortStrategy sort)
    {
        this.sortStrategy = sort;
    }
}
```

SortStrategy 类扮演了策略模式中的抽象策略角色。在这里，这个抽象角色规定出具体策略角色所必须实现的接口，即 sort()方法，如代码清单 9 所示。

代码清单 9：抽象角色 SortStrategy 的源代码

```
abstract public class SortStrategy
{
    /**
     *  策略方法
     */
     public abstract void sort();
}
```

BinSort 类以及 BubbleSort、HeapSort、QuickSort、RadixSort 等类均是具体策略角色的扮演者。BinSort 类的源代码如代码清单 10 所示。

代码清单 10：具体策略类 BinSort（二元排序）的代码

```
public class BinSort extends SortStrategy
{
    /**
     *  策略方法
     */
     public void sort()
     {
         //sorting logic here
     }
}
```

下面是具体策略角色之一——BubbleSort 类的源代码，如代码清单 11 所示。

代码清单 11：策略类 BubbleSort（泡沫排序）的源代码

```
public class BubbleSort extends SortStrategy
{
    /**
     *  策略方法
     */
     public void sort()
     {
         //sorting logic here
     }
}
```



下面是具体策略角色之一——HeapSort 类的源代码，如代码清单 12 所示。

代码清单 12：策略类 HeapSort（堆栈排序）的源代码

```
public class HeapSort extends SortStrategy
{
    /**
     *  策略方法
     */
     public void sort()
     {
         //sorting logic here
     }
}
```

下面是具体策略角色之一——QuickSort 类的源代码，如代码清单 13 所示。

代码清单 13：策略类 QuickSort（快速排序）的源代码

```
public class QuickSort extends SortStrategy
{
    /**
     *  策略方法
     */
     public void sort()
     {
         //sorting logic here
     }
}
```

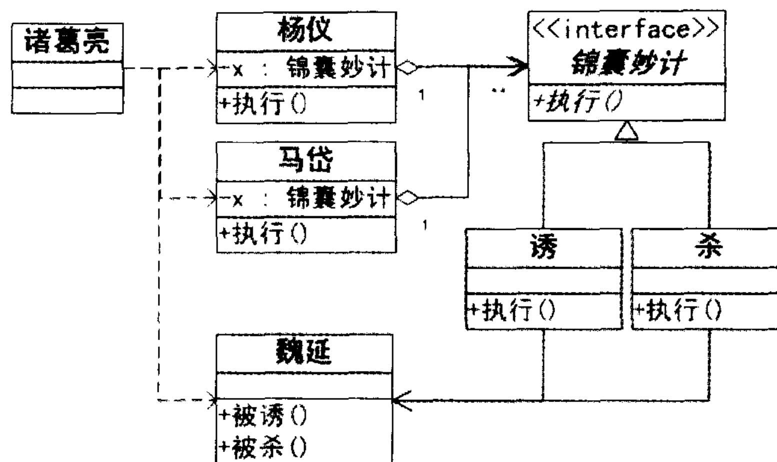
下面是具体策略角色之一——RadixSort 类的源代码，如代码清单 14 所示。

代码清单 14：具体策略类 RadixSort（基数排序）的源代码

```
public class RadixSort extends SortStrategy
{
    /**
     *  策略方法
     */
     public void sort()
     {
         //sorting logic here
     }
}
```

2. 这不是策略模式。策略模式的用意在于处理多于一个的算法时把算法和使用算法的客户端分割开。策略模式把每一个算法包装在具体策略对象里面，所有的具体策略模式都需要实现相同的接口。

在这里看不到这样的用意，因此与策略模式无关。因此在历史上同样叫做“锦囊妙计”的故事，可能属于不同的模式。命令模式更接近这个历史故事所描述的，如下图所示。



不熟悉命令模式的读者请阅读本书“命令模式”一章。

参考文献

[JMOR] John Morris. Programming Languages and Data Structures. http://ciips.ee.uwa.edu.au/~morris/

第 36 章　模版方法（Template Method）模式

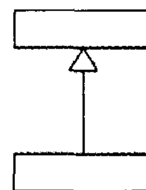
模版方法模式是类的行为模式[GOF95]。准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模版方法模式的用意。

36.1　引　　言

很多人可能没有想到，模版方法模式实际上是本书所讨论的所有模式中最为常见的几个模式之一，而且很多读者可能使用过模版方法模式而没有意识到自己已经使用了这个模式。模版方法模式是基于继承的代码复用的基本技术，模版方法模式的结构和用法也是面向对象设计的核心。

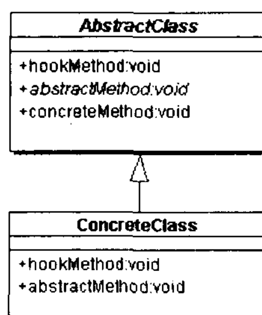
模版方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法（primitive method）；而将这些基本法方法总汇起来的方法叫做模版方法（template method），这个设计模式的名字就是从此而来。

模版方法所代表的行为称为顶级行为，其逻辑称为顶级逻辑。模版方法模式的简略类图如右图所示。



36.2　模版方法模式的结构

模版方法模式的静态结构如下图所示。



这里涉及到两个角色：

抽象模版（Abstract Template）角色有如下的责任：

- 定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。
- 定义并实现了一个模版方法。这个模版方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

具体模版（Concrete Template）角色有如下的责任：

- 实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。
- 每一个抽象模版角色都可以有任意多个具体模版角色与之对应，而每一个具体模版角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

本节在下面给出示意性源代码。

首先抽象模版角色提供了一个具体方法 TemplateMethod()，此具体方法调用一些抽象方法，包括 doOperation1()，doOperation2()等，如代码清单 1 所示。

代码清单 1：抽象模版类的示意性源代码

```
abstract public class AbstractClass
{
    /**
     * 模版方法的声明和实现
     */
    public void TemplateMethod()
    {
        // 调用基本方法（由子类实现）
        doOperation1();
        // 调用基本方法（由子类实现）
        doOperation2();
        // 调用基本方法（已经实现）
        doOperation3();
    }
    /**
     * 基本方法的声明（由子类实现）
     */
    protected abstract void doOperation1();
    /**
     * 基本方法的声明（由子类实现）
     */
    protected abstract void doOperation2();
    /**
     * 基本方法（已经实现）
     */
    private final void doOperation3()
    {
```



```
            //do something
        }
    }
```

显然 doOperation1()，doOperation2()等基本方法是顶级逻辑的组成步骤，这个顶级逻辑由 TemplateMethod()方法代表。显然，抽象模版类自己并不给出这些基本方法的实现，而是把这些基本方法交给子类去实现。

具体模版角色负责实现抽象方法，如代码清单 2 所示。

代码清单 2：具体模版类的示意性源代码

```
public class ConcreteClass extends AbstractClass
{
    /**
     * 基本方法的实现
     */
    public void doOperation1()
    {
        System.out.println("doOperation1();");
    }
    /**
     * 基本方法的实现
     */
    public void doOperation2()
    {
        //像下面这样的调用不应当发生
        //doOperation3();
        System.out.println("doOperation2();");
    }
}
```

这个具体类实现了父类所声明的基本方法——doOperation1()和 doOperation2()，而这两个基本方法所代表的就是强制子类实现的剩余逻辑。

36.3 "好莱坞原则"

在好莱坞工作的演艺界人士都了解，在把简历递交给好莱坞的娱乐公司以后，所能做的就是等待。这些公司会告诉他们"不要给我们打电话，我们会给你打"。这便是所谓的"好莱坞原则"。好莱坞原则的关键之处，是娱乐公司对娱乐项目的完全控制。应聘的演艺人员只是被动地服从总项目流程的安排，在需要的时候完成流程中的一个具体环节。

虽然"好莱坞原则"很早的时候在不同的题材里被讨论过[SWE85]，使用这个比喻描写模版方法模式则是由[GoF]给出的，他们认为"好莱坞原则"体现了模版模式的关键：子类可以置换掉父类的可变部分，但是子类却不可以改变模版方法所代表的顶级逻辑。

每当定义一个新的子类时，不要按照控制流程的思路去想，而应当按照"责任"的思

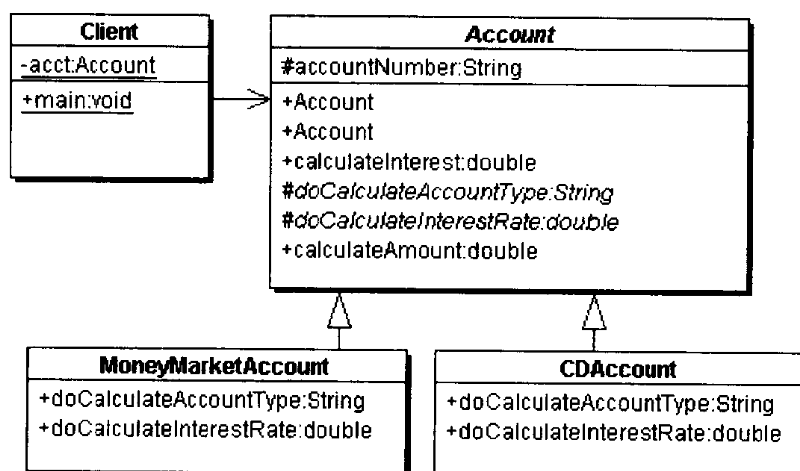
路去想。换言之，应当考虑有哪些操作是必须置换掉的，哪些操作是可以置换掉的，以及哪些操作是不可以置换掉的。使用模版方法模式可以使这些责任变得清晰。

36.4 一个例子

考虑一个计算存款利息的例子。假设系统需要支持两种存款账号，即货币市场（Money Market）账号和定期存款（CD 或 Certificate of Deposite）账号。这两种账号的存款利息是不同的，因此，在计算一个存户的存款利息额时，必须区分两种不同的账号类型。

这个系统的总行为应当是计算出利息，这也就决定了作为一个模版方法模式的顶级逻辑应当是利息计算。由于利息计算涉及到两个步骤：一是确定账户的类型，二是确定利息的百分比，因此系统需要两个基本方法：一个基本方法给出账号种类，另一个基本方法给出利息百分比。这两个基本方法构成具体逻辑，因为账号的类型不同，所以具体逻辑会有所不同。

显然，系统需要一个抽象角色给出顶级行为的实现，而将两个作为细节步骤的基本方法留给具体子类实现。由于需要考虑的账号有两种：一是货币市场账号，二是定期存款账号，因此如果假设系统为一个 Client 对象所调用的话，系统的类图结构将如下图所示。



模版方法模式的实现方法是从上到下的，也就是说，需要首先给出顶级的逻辑，然后给出具体步骤的逻辑。因此，本系统不可避免地要从抽象模版角色开始设计。这个角色的源代码如代码清单 3 所示。

代码清单 3：抽象类 Account 的源代码

```
package com.javapatterns.templatemethod.InterestRate;
abstract public class Account
{
    protected String accountNumber;
    /**
     *  默认构造子
     */
```

```
    public Account()
    {
        accountNumber = null;
    }
    /**
     *  构造子
     */
    public Account(String accountNumber)
    {
        this.accountNumber = accountNumber;
    }
    /**
     *  模版方法，计算利息数额
     */
    final public double calculateInterest()
    {
        double interestRate = doCalculateInterestRate();
        String accountType = doCalculateAccountType();
        double amount = calculateAmount(accountType,
            accountNumber);
        return amount * interestRate;
    }

    /**
     *  基本方法留给子类实现
     */
    abstract protected String doCalculateAccountType() ;
    /**
     *  基本方法留给子类实现
     */
    abstract protected double doCalculateInterestRate() ;
    /**
     *  基本方法，已经实现
     */
    final public double calculateAmount(String accountType,
        String accountNumber)
    {
        //retrieve amount from database...
        return 7243.00D;
    }
}
```

其中，calculateAmount()方法应当连接数据库并取出数据库中的数据，但是本章不打算在这里涉及与模式的讨论没有太大关系的细节，因此使用了直接返还一个确定数额的示意性做法。

显然，这个抽象角色给出了顶级逻辑的骨架，即 calculateInterest()的内容，只是将内

容的具体步骤委派给不同的基本方法，在这里分别是 doCalculateAccountType()和 doCalculateInterestRate()，而这两个抽象方法则留给具体子类实现，如代码清单 4 所示。

代码清单 4：具体子类 MoneyMarketAccount 的源代码

```
package com.javapatterns.templatemethod.InterestRate;
public class MoneyMarketAccount extends Account
{
    /**
     *  基本方法在这里实现
     */
    public String doCalculateAccountType()
    {
        return "Money Market";
    }
    /**
     *  基本方法在这里实现
     */
    public double doCalculateInterestRate()
    {
        return 0.045D;
    }
}
```

在上面的 MoneyMarketAccount 子类中，基本方法 doCalculateAccountType()和 doCalculateInterestRate()分别被实现。具体子类 CDAccount 如代码清单 5 所示。

代码清单 5：具体子类 CDAccount 的源代码

```
package com.javapatterns.templatemethod.InterestRate;
public class CDAccount extends Account
{
    /**
     *  基本方法在这里实现
     */
    public String doCalculateAccountType()
    {
        return "Certificate of Deposite";
    }
    /**
     *  基本方法在这里实现
     */
    public double doCalculateInterestRate()
    {
        return 0.065D;
    }
}
```

同样地，在上面的 CDAccount 子类中，基本方法 doCalculateAccountType()和



doCalculateInterestRate()分别被实现。客户端 Client 的源代码如代码清单 6 所示。

代码清单 6：客户端 Client 的源代码

```
package com.javapatterns.templatemethod.InterestRate;
public class Client
{
    private static Account acct = null;
    public static void main(String[] args)
    {
        acct = new MoneyMarketAccount();
        System.out.println("Interest from Money Market account "
            + acct.calculateInterest());
        acct = new CDAccount();
        System.out.println("Interest from CD account "
            + acct.calculateInterest());
    }
}
```

上面的客户端 Client 的 main()方法调用了真实类型为 MoneyMarketAccoun 和 CDAccount 的两个对象的模版方法 calculateInterest()，从而得到下面的结果，如代码清单 7 所示。

代码清单 7：运行的结果

```
Interested from Money Market account 325.935
Interested from CD account 470.795
```

显然，货币市场账号和定期存款账号的不同利息数额是由于基本方法在不同的具体子类中有不同的实现所造成的。

36.5　继承作为复用的工具

在阅读本节之前，建议读者首先阅读本书的“抽象类”一章。

使用继承作为复用的手段必须慎重，Java 语言的设计师对使用继承作为复用的工具有着不同层次上的认识。

不知其一

首先，初学 Java 的程序员可能不知道什么是继承，或者认为“继承”是高深的工具。那时候，大部分的功能复用都是通过委派进行的。

知其一、不知其二

然后慢慢地，他们发现在 Java 语言里实现继承并不困难，并且初步认识到继承可以

使子类一下子得到基类的行为。这时他们就会跃跃欲试了，试图使用继承作为功能复用的主要工具，并把原来应当使用委派的地方，改为使用继承，这时继承就有被滥用的危险。

知其二

很多面向对象的设计专家从 1986 年就开始警告继承关系被滥用的可能。有一些面向对象的编程语言，如 SELF 语言，甚至将类的继承关系从语言的功能中取消掉，改为完全使用委派。

其他的设计师虽然不提倡彻底取消继承，但无一例外地鼓励在设计中尽可能使用委派关系代替继承关系。比如在[GOF95]一书中，状态模式、策略模式、装饰模式、桥梁模式以及抽象工厂模式均是将依赖于继承的实现转换为基于对象的组合和聚合的实现，这些模式的要点就是使用委派关系代替继承关系。

知其三

是不是继承就根本不该使用呢？事实上对数据的抽象化、继承、封装和多态性并称 Java 和其他绝大多数的面向对象语言的几项最重要的特性。继承不应当被滥用，并不意味着继承根本就不该使用。因为继承容易被滥用就彻底抛弃继承，无异于因噎废食。

继承使得类型的等级结构易于理解、维护和扩展，而类型的等级结构非常适合于抽象化的设计、实现和复用。尽管[GOF95]所给出的设计模式基本上没有太多基于继承的模式，很多模式都是用继承的办法定义、实现接口的。多数的设计模式都描写一个以抽象类作为基类，以具体类作为实现的等级结构，比如适配器模式、合成模式、桥梁模式、状态模式等。

模版方法模式则更进了一步：此模式鼓励恰当地使用继承。此模式可以用来改写一些拥有相同功能的相关的类，将可复用的一般性的行为代码移到基类里面，而把特殊化的行为代码移到子类里面。

因此，熟悉模版方法模式便成为一个重新学习继承的好地方。

36.6 Java 语言里面使用过的模版方法模式

Java 语言在很多地方使用过模版方法模式，这里给出几个例子。

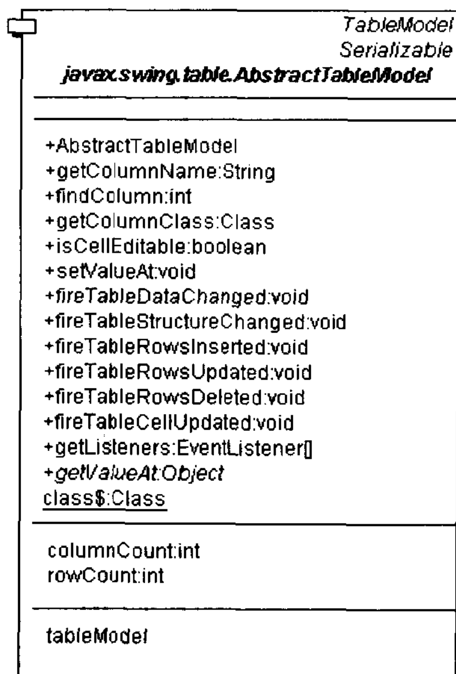
HttpServlet 技术

作为 Java 对 Web 系统的解决方案，HttpServlet 技术是建立在模版方法模式的基础之上的。HttpServlet 类提供了一个 service()方法。这个方法调用 7 个 do 方法中的一个或几个，完成对客户端调用的处理。这些 do 方法则要由具体 HttpServlet 类提供。在这里，service()方法便是模版方法，7 个 do 方法便是基本方法。关于这一应用的讨论，请见下一章“专

题：模版方法模式与 Servlet”。

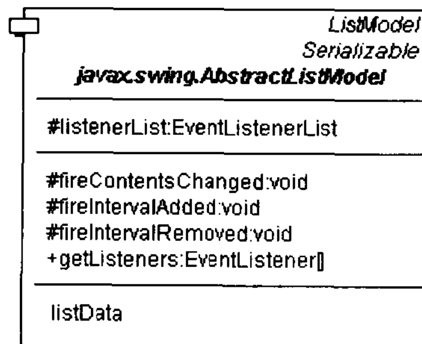
AbstractTableModel 类

这个抽象类实现了 TableModel 接口所要求的所有方法，是用来处理 Swing JTable 构件的显示功能的。AbstractTableModel 类为除去三个方法的所有方法提供了默认实现，因此当一个系统准备一个 AbstractTableModel 类的数据子类时，就需要提供这三个方法的具体实现，如下图所示。



AbstractListModel 类

与 AbstractTableModel 类一样，AbstractListModel 抽象类提供了一个 ListModel 接口的默认实现，通过扩展 AbstractListModel 抽象类可以建立一个系统的数据模型。这个抽象类是一个模版方法模式的应用，如下图所示。



36.7 模版方法模式中的方法

模版方法中的方法可以分为两大类：模版方法（Template Method）和基本方法（Primitive Method）。

模版方法

一个模版方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或一个总行为的方法。这个模版方法一般会在抽象类中定义，并由子类不加以修改地完全继承下来。

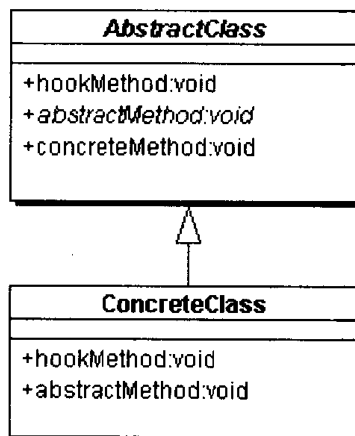
一个抽象类可以有任意多个模版方法，而不限于一个。每一个模版方法都可以调用任意多个具体方法。

基本方法

基本方法又可以分为三种：抽象方法（Abstract Method）、具体方法（Concrete Method）和钩子方法（Hook Method）。

- 抽象方法：一个抽象方法由抽象类声明，由具体子类实现。在 Java 语言里一个抽象方法以 abstract 关键字标示出来。
- 具体方法：一个具体方法由抽象类声明并实现，而子类并不实现或置换。有些具体方法可以起到工厂方法的作用，这样的具体方法又叫做工厂方法。在 Java 语言里面，一个具体方法没有 abstract 关键字。
- 钩子方法：一个钩子方法由抽象类声明并实现，而子类会加以扩展。通常抽象类给出的实现是一个空实现，作为方法的默认实现。

在下面类图给出的例子里，AbstractClass 是一个抽象类，它带有三个方法。其中 abstractMethod()是一个抽象方法，它由抽象类声明为抽象方法，并由子类提供实现；concreteMethod()是一个具体方法，它由抽象类声明并实现；hookMethod()是一个钩子方法，它由抽象类声明并提供默认实现，并且由子类置换掉，如右图所示。



如何决定一个基本方法是哪一种

假设给出一个应当使用模版方法模式的问题以及一系列的基本方法，怎样才能决定每一个基本方法应当是哪一个种类的基本方法呢？

决定哪一个方法应作为具体方法是一个较为简单的任务，而决定哪一个方法作为抽象



方法而不是钩子方法，或者应作为钩子方法而不是抽象方法，则常常不是一个容易的任务。

模版方法模式的设计理念是尽量减少必须由子类置换掉（override）的基本方法的数目。

默认钩子方法

正如上面所谈到的，一个钩子方法常常由抽象类给出一个空实现作为此方法的默认实现。这种空的钩子方法叫做“Do Nothing Hook”。显然，这种默认钩子方法在缺省适配模式里已经见过了。一个缺省适配模式讲的是一个类为一个接口提供一个默认的空实现，从而使得缺省适配类的子类不必像实现接口那样必须给出所有方法的实现，因为通常一个具体类并不需要所有的方法。

命名规则

命名规则是设计师之间赖以沟通的管道之一，使用恰当的命名规则可以帮助不同设计师之间的沟通。

钩子方法的名字应当以 do 开始，这是熟悉设计模式的 Java 程序设计师的标准做法，本书推荐使用模版方法模式的读者使用这一命名规则。正如在上面的例子里面，两个基本方法分别是 doCalculateAccountType()和 doCalculateInterestRate()；在 HttpServlet 里面，基本方法也遵从这一命名规则。

36.8　模版方法模式在代码重构中的应用

模版方法模式可以作为方法层次上的代码重构（Code Refactor）的一个重要手段。

一个初级的程序员常常会将很多的代码放在一个很大的方法里面，造成一个几千行的大方法。这样的方法应当拆分成一些较小的方法，拆分的策略往往可以使用模版方法模式。

将大方法打破

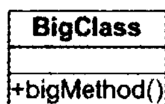
假设下面就是这个需要重构的过大的方法，如代码清单 8 所示。

代码清单 8：需要重构的源代码

```
public void bigMethod()
{
    ...
    代码块 1
    ...
    代码块 2
    ...
    代码块 3
```

```
    ...
    代码块 4
    ...
    代码块 5
    ...
}
```

其类图如下图所示。



首先，将这个大方法作为模版方法。将这个大方法打碎成为很多的步骤，每一个步骤的代码划分到一个新的小方法里面，由模版方法调用一个个的小方法，如代码清单 9 所示。

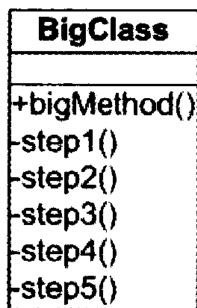
代码清单 9：初步重构后的源代码

```
public void bigMethod()
{
    step1();
    step2();
    if (...)
    {
        step3();
    }
    else if (...)
    {
        step4();
    }
    else
    {
        step5();
    }
}
private void step1()
{
    原来的代码块 1
}
private void step2()
{
    原来的代码块 2
}
private void step3()
{
    原来的代码块 3
}
private void step4()
```



```
{
    原来的代码块 4
}
private void step5()
{
    原来的代码块 5
}
```

划分后的类图如下图所示。



建立取值方法

在这样做了之后，一个需要解决的问题就是那些原来在大方法内部声明的属性。这些属性原本是局域变量。为了能将这些变量与基本方法共享，可以有两种做法：一是将它们改为超类层次上的私有或保护变量；二是建立一些抽象取值方法，而将取值方法的实现推迟给子类。也就是说，每一个变量都封装在一个方法里面，这个方法所做的惟一的事情就是返还这个变量。

这样做的好处，是可以将状态的声明尽量推迟到子类，而使得等级结构的高端与状态无关。

建立常量方法

与属性一样，常量也应当以常量取值方法的形式封装起来，这个常量方法所做的惟一一件事情就是返还这个常量。与可变的状态属性一样，常量方法可以将常量的声明推迟到子类中。

如此反复

如此反复直到所有的基本方法都变成基本上一样的令人满意的细粒度（granularity），而且所有的常数都放到了常数方法里面。

这时候就得到了一个类，里面有一个模版方法和一系列的基本方法。一般来说，到此结束重构是可以的了。但是，如果这些方法里面有一些特征相同、功能相同而细节不同的方法，那么重构过程就可以利用继承和继承带来的多态性继续下去。

以多态性取代条件转移

需要注意的是，在现在的 bigMethod()方法中有一个条件转移块，这样的条件转移在大而不当的方法中常常见到，而在这里仅仅给出一个例子。这里使用到的另一个重要的重构原则便是以多态性代替条件转移块。

将所考虑的类作为模版方法模式的抽象类，设计一些具体子类，再将基本方法中特征相同、功能相同而细节不同的基本方法划分到不同的具体子类里面去。假设方法 step3()，step4()和 step5()是特征相同、功能相同而实现不同的方法，那么这三个方法可以作为一个新方法 newMethod()的多态性体现，划分到三个不同的子类中去，如代码清单 10 所示。

代码清单 10：重构的最后结果

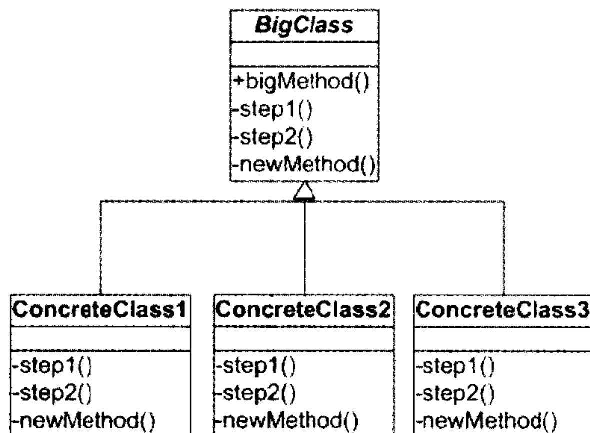
```
public abstract class AbstractClass
{
    public void bigMethod()
    {
        step1();
        step2();
        newMethod();
    }
    protected abstract void step1();
    protected abstract void step2();
    protected abstract void newMethod();
}
public class ConcreteClass1 extends AbstractClass
{
    public void newMethod()
    {
        原来的代码块 3
    }
}
public class ConcreteClass2 extends AbstractClass
{
    public void newMethod()
    {
        原来的代码块 4
    }
}
public class ConcreteClass3 extends AbstractClass
{
    public void newMethod()
    {
        原来的代码块 5
    }
}
```



实施委派

在需要的情况下，建立独立的类负责独立的行为，从而可以将独立的行为委派到独立的对象里面。

下图所示就是代码重构后的最终结果。



可以看出，新的 BigClass 便是一个抽象模版角色，而 ConcreteClass1，ConcreteClass2 和 ConcreteClass3 便是具体模版角色。在抽象模版角色里，有一个顶级的逻辑 bigMethod()。

36.9　重构的原则

本书在“抽象类”一节中曾经谈到，在对一个继承的等级结构做重构时，一个应当遵从的原则便是将行为尽量移动到结构的高端，而将状态尽量移动到结构的低端。本节将继续这个讨论，并将这个重构原则与模版方法模式结合起来。

1995 年，Auer 曾在文献[AUER95]中指出：

（1）应当根据行为而不是状态定义一个类。也就是说，一个类的实现首先建立在行为的基础之上，而不是建立在状态的基础之上。

（2）在实现行为时，是用抽象状态而不是用具体状态。如果一个行为涉及到对象的状态时，使用间接的引用而不是直接的引用。换言之，应当使用取值方法而不是直接引用属性。

（3）给操作划分层次。一个类的行为应当放到一个小组核心方法（kernel methods）里面，这些方法可以很方便地在子类中加以置换。

（4）将状态属性的确认推迟到子类中。不要在抽象类中过早地声明属性变量，应将它们尽量地推迟到子类中去声明。在抽象超类中，如果需要状态属性的话，可以调用抽象的取值方法，而将抽象的取值方法的实现放到具体子类中。

如果能够遵从这样的原则，那么就可以在等级结构中将接口与实现分隔开来，将抽象与具体分割开来，从而保证代码可以最大限度地被复用。

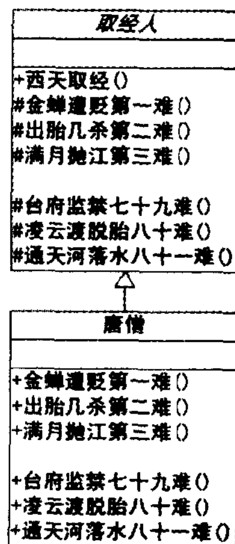
实际上读者可以看出，在这样的等级结构里面，一个抽象超类给出结构的顶级行为，而由一个或者多个具体子类具体实现顶级行为的细节步骤。子类可以容易地置换掉超类的方法，而不需要继承超类的状态。

这个过程实际上是将设计师引导到模版方法模式上去。模版方法模式的基本方法便是Auer的核心方法（kernel methods），而模版方法便是使用核心方法的顶级方法。

36.10 西天取经的八十一难

无独有偶，在《西游记》中，唐僧到西天取经途中所历的九九八十一难均是菩萨所控制的一个项目，菩萨管理项目的方法与好莱坞控制娱乐项目的方式并无二致。本节就以设计模式的观点加以分析，以娱读者。

当年五方揭谛、四值功曹、六丁六甲、护教伽蓝暗中保护唐僧到了西天取到真经，向菩萨缴回法旨。菩萨查了查唐僧师徒的受难簿，突然发现唐僧少历了一难，急声传道："佛门中九九归真，圣僧受过八十难，还少一难，不得完成此数。"即令揭谛，"赶上金刚，还生一难者。"于是唐僧又历了一难。



因此，唐僧一共经历了九九八十一难，而这八十一难全都是在五方揭谛、四值功曹、六丁六甲、护教伽蓝暗中引导下经历的。换言之，菩萨在事先制定了一个顶级逻辑框架，而将逻辑的细节留给具体子类（在这里就是"唐僧"类）去实现，如右图所示。

由此可见，菩萨当年就是使用模版方法模式处理唐僧去西天取经的项目安排的。在这个模拟系统中，关键性的角色就是抽象类"取经人"，这个类的模拟源代码如代码清单11所示。

代码清单 11：抽象模版类的示意性源代码

```
abstract public class 取经人
{
    /**
     * 模版方法的声明和实现
     */
    public void 西天取经()
    {
        // 调用基本方法（由子类实现）
        金蝉遭贬第一难();
        // 调用基本方法（由子类实现）
        出胎几杀第二难();
        // 调用基本方法（由子类实现）
        满月抛江第三难();
        ...
    }
    /**
```

```
     *  基本方法的声明（由子类实现）
     */
    protected abstract void 金蝉遭贬第一难();
    /**
     *  基本方法的声明（由子类实现）
     */
    protected abstract void 出胎几杀第二难();
    /**
     *  基本方法的声明（由子类实现）
     */
    protected abstract void 满月抛江第三难();
    ...
}
```

显然，这个抽象类有一个模版方法，就是“西天取经()”方法，这个方法代表了抽象类的顶级逻辑。“西天取经()”方法不断调用基本方法，分别是“金蝉遭贬第一难()”、“出胎几杀第二难()”、“满月抛江第三难()”等八十一个抽象方法。这些抽象方法是强制要求具体子类“唐僧”来实现的，因此代表了抽象类的剩余逻辑。

读者可以看出，这是一个模版方法模式的应用实例。

问答题

请给本章 AbstractListModel 类图中的两个类提供示意性源代码。

问答题答案

抽象类 AbstractClass 的示意性源代码如代码清单 12 所示。

代码清单 12：AbstractClass 的示意性源代码

```
package com.javapatterns.templatemethod.hook;
abstract public class AbstractClass
{
    /**
     *  钩子方法的声明
     */
    public void hookMethod()
    {}
    /**
     *  抽象方法的声明
     */
    public abstract void abstractMethod();
    /**
     *  具体方法
     */
    public void concreteMethod()
```

```
    {
        System.out.println("A concrete method.");
    }
}
```

可以看出，上面的钩子方法 hookMethod()只给出了一个默认实现。在下面的具体子类 ConcreteClass 中，hookMethod()方法被重新实现，如代码清单 13 所示。

代码清单 13：ConcreteClass 的示意性源代码

```
package com.javapatterns.templatemethod.hook;
public class ConcreteClass extends AbstractClass
{
    /**
     * 钩子方法的实现
     */
    public void hookMethod()
    {
        System.out.println("A re-implemented hook method.");
    }
    /**
     * 抽象方法
     */
    public void abstractMethod()
    {
        System.out.println("An abstract method.");
    }
}
```

参考文献

[SWE85] Richard E. Sweet. The Mesa Programming Environment. SIGPLAN Notices, 20 (7): 216-229, July 1985

[Vlissides96] John Vlissides. C++ Report. Pattern Hatching, column for February '96 issue

[FOWLER00] Martin Fowler . Refactoring – Improving the Design of Existing Code. Addison-Wesley, 2000

[AUER95] Ken Auer. Reusability through self-encapsulation. in J. O. Coplien & D. C. Schmidt (eds.), Pattern Languages of Program Design, p 505-516, Addision-Wesley, 1995

第 37 章 专题：Servlet 技术中的模式

本章所讨论的 Servlet 技术基于 Sun Microsystem 所提供的 Servlet 2.3 版本。这一版本是作为 Servlet 技术的最终版本提出来的，当然它是否真的能够成为最终版，还需要时间的检验。如果读者没有这一版的 Servlet 类的源代码文件，可以从 Sun Microsystem 的网站下载。

本书建议读者在阅读本章之前，首先熟悉 Java Servlet 技术，并阅读"模版方法（Template Method）模式"和"观察者（Observer）模式"两章。

37.1 Servlet 技术介绍

Java 的 Servlet 提供了 J2EE 技术的 Web 构件的基础，它提供了 Servlet、JSP（Java ServerPages）和 EJB（Enterprise JavaBeans）等架构。EJB 所提供的技术在应用开发环境中所起的作用是与其他两者不同的，但是 Servlet 和 JSP 所起的作用则是很相近的。

客户可以对一个 J2EE 服务器提出请求，一个 Servlet 可以对客户的请求做出反应。最通常的请求方式便是 HTTP 请求，这种请求通过用户的浏览器传到 Web 服务器和 J2EE 服务器上。因此，用户在与一个网页发生相互作用时，可以通过 Web 服务器和 J2EE 服务器与 Servlet 相互作用。而 Servlet 可以对用户的请求做出响应，针对请求提供相应的操作，包括通过 JDBC 与数据库发生相互作用等。

Servlet 的最广泛用途便是建造动态的网页。没有 Servlet 或相应的动态技术（如 CGI，Cold Fusion 或 ASP），一个 Web 服务器只能提供静态的网页，不能针对用户的请求提供相应的服务器端的操作，如查询数据库并根据查询结果构造一个新的结果网页等。

不错，在 J2EE 技术出现之前就已经有相应的动态网页生成技术，如 CGI、Cold Fusion 或 ASP 等。但是与这些技术相比，J2EE 有更好的系统伸缩性（scalability），非常自恰的系统模型，完全对象化的模型等。因此，J2EE 有着自己的优越性。

Servlet 的两个库包

所有的 Servlet 必须遵循 Java Servlet API。这个 API 是由两个库包及一系列类组成的：

- javax.Servlet：这个库包提供了普通 Servlet 模型。
- javax.Servlet.http：这个库包提供了 HTTP 和 HTTPS 的 Servlet 模型。

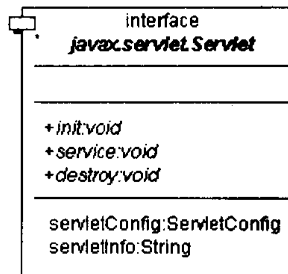
普通 Servlet 模型只是假定协议是建立在 TCP/IP 之上的，并没有对请求-回答（request-response）的协议（protocol）做出任何具体的假设，所有与 HTTP 和 HTTPS 有关的逻辑均在 javax.Servlet.http 库里面。这种办法可以在最大程度上把可能发生变化的部分与一般性逻辑分开，这也就是"开-闭"原则和"封装变化"原则的具体体现（关于"开

–闭”原则和“封装变化”原则的详细讨论，请见本书“开–闭原则（OCP）”一章。

Servlet 的主要的类

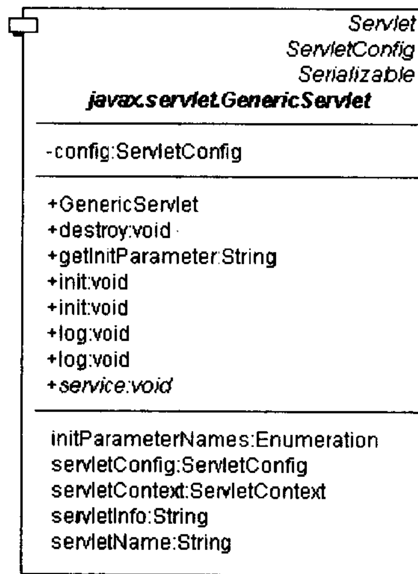
Servlet 库中有三个类提供了主要的 Servlet 技术架构，这三个类是：Servlet 接口、GenericServlet 类和 HttpServlet 类。

首先来看一看 Servlet 接口，如右图所示。

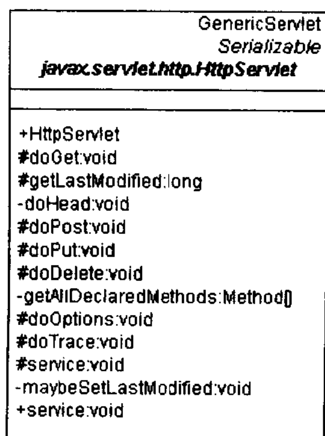


Servlet 接口声明了三个方法：init()，destroy()和 service()方法。这三个方法决定了 Servlet 的生命周期。

其次是 GenericServlet 类，如下图所示。



GenericServlet 是一个抽象类，它提供了 Servlet 接口的默认实现，但是却留下了一个 service()方法，这个方法仍然是一个抽象方法。任何具体的 Servlet 类均必须提供 service()方法，以便提供必要的具体行为。



最后是 HttpServlet 类，如左图所示。

HttpServlet 类继承自 GenericServlet。虽然它本身也是一个抽象类，但是它为所有行为包括 service()方法和七个 do 方法都提供了默认实现：

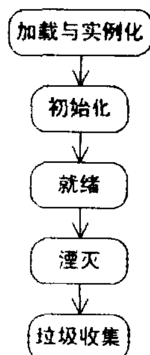
- goGet()
- doHead()
- doPost()
- doPut()
- doDelete()
- doOption()
- doTrace()

这七个 do 方法基本涵盖了 HTTP 协议的各种请求的种类。



HttpServlet 的生命周期

Servlet 的生命周期描述一个 Servlet 如何被加载和被初始化，它怎样接收请求、响应请求以提供服务。Java 的 HttpServlet 类必须在一个 Servlet 引擎所提供的容器环境里面运行，因此，它的生命周期是由容器环境管理的。



Servlet 生命周期由接口 javax.Servlet.Servlet 所定义。所有的 Java Servlet 必须直接或间接地实现 javax.Servlet.Servlet 接口，才能在 Servlet 容器环境里面运行。

右图所示的状态图表明了 HttpServlet 的生命周期。

加载

Servlet 容器可以同时创建一个或多个 Servlet 对象。这个过程可以在 Servlet 容器加载 Servlet 类时执行，或者在 Servlet 对象响应请求时执行，也可以在两者之间的任何时候执行。

初始化

Servlet 容器调用一个 Servlet 的 init()方法将此 Servlet 初始化。初始化时，Servlet 对象可以从数据库里读取初始数据，建立 JDBC Connection，或者建立对其他有价值的资源的引用等。

就绪

Servlet 被初始化以后，处于能响应请求的就绪（Ready）状态。当客户端有一个请求时，Servlet 容器将 ServletRequest 和 ServletResponse 对象都转发给 Servlet，这两个对象以参数的形式传给 service()方法。

这样 service()方法会进一步调用 doGet()和 doPost()等方法处理相应的 HTTP 请求。

释放和湮灭

调用 destroy()方法，系统将释放相应的资源，所有失去引用的 Java 对象都会被垃圾收集器收集。

37.2　模版方法模式的使用

Servlet 在几个地方采用了模版方法模式的设计。

service()方法是一个模版方法

首先，HttpServlet 类提供了一个 service()方法，这个方法调用七个 do 方法中的一个或几个，完成对客户端调用的响应。这些 do 方法需要由 HttpServlet 的具体子类提供，因此这是典型的模版方法模式。下面是默认的 service()方法的源代码，如代码清单 1 所示。

代码清单 1：service()方法的源代码

```
protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    String method = req.getMethod();
    if(method.equals("GET"))
    {
        long lastModified = getLastModified(req);
        if(lastModified == -1L)
        {
            doGet(req, resp);
        }
        else
        {
            long ifModifiedSince = req.getDateHeader("If-Modified-Since");
            if(ifModifiedSince == -1L)
            {
                doGet(req, resp);
            }
            else
            {
                maybeSetLastModified(resp, lastModified);
                long now = System.currentTimeMillis();
                if(now < ifModifiedSince || ifModifiedSince < lastModified)
                    doGet(req, resp);
                else
                    resp.sendError(304);
            }
        }
    }
    else
    {
        if(method.equals("HEAD"))
        {
            long lastModified = getLastModified(req);
            maybeSetLastModified(resp, lastModified);
            doHead(req, resp);
        }
        else
        {
            if(method.equals("POST"))
            {
                doPost(req, resp);
            }
        }
        else
```



```
        {
            if(method.equals("PUT"))
            {
                doPut(req, resp);
            }
        }
        else
        {
            if(method.equals("DELETE"))
            {
                doDelete(req, resp);
            }
        }
        else
        {
            if(method.equals("OPTIONS"))
            {
                doOptions(req, resp);
            }
        }
        else
        {
            if(method.equals("TRACE"))
            {
                doTrace(req, resp);
            }
            else
            {
                String errMsg = lStrings.getString("http.method_not_implemented");
                Object errArgs[] = new Object[1];
                errArgs[0] = method;
                errMsg = MessageFormat.format(errMsg, errArgs);
                resp.sendError(501, errMsg);
            }
        }
    }
}
```

当然，这个 service()方法可以被子类置换掉。

标识接口

不难看出，java.util.EventListener 接口没有任何的方法，它是一个标识接口，如代码清单 2 所示。

代码清单 2：EventListener 接口的源代码

```
package java.util;
public interface EventListener
{
}
```

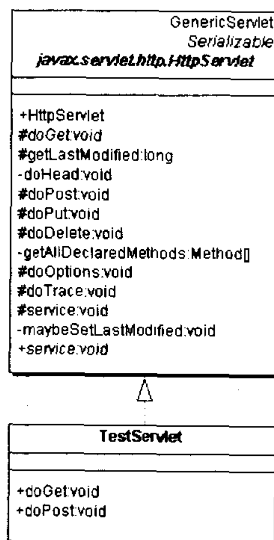
一个例子

本节给出一个简单的 Servlet 的例子，这个 Servlet 在被调用时给出一个 HTML 信息，包括两行字：

第一行字根据 HTTP 请求是 GET 还是 POST 打印出：“The http request is GET”或者“The http request is POST”。

第二行是当前的时间。

由于 Servlet 必须能够区分 HTTP 请求的类型是 GET 还是 POST，因此系统需要分别实现 doGet()和 doPost()两种 do 方法。这个简单系统的类图如下图所示。



从上面的类图可以看出，新的 Servlet 类 TestServlet 是 HttpServlet 的子类，并且置换掉了父类 HttpServlet 的两个方法：doGet()和 doPost()。下面是这个新的 Servlet 类 TestServlet 的源代码，如代码清单 3 所示。

代码清单 3：TestServlet 类的源代码

```
package com.javapatterns.Servlet;
import java.util.Date;
import java.io.*;
import javax.Servlet.*;
import javax.Servlet.http.*;
public class TestServlet extends HttpServlet
{
    /**
```



```
    *    当 HTTP 请求是 GET 时，此方法会被触发
    */
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<h1>The http request is GET.</h1>");
        out.println("<h2>Now is " + new Date() + "</h2>");
        out.close();
    }
   /**
    *    当 HTTP 请求是 POST 时，此方法会被触发
    */
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<h1>The http request is POST.</h1>");
        out.println("<h2>Now is " + new Date() + "</h2>");
        out.close();
    }
}
```

37.3　观察者模式的应用

关于观察者模式的知识，请读者阅读本书“观察者（Observer）模式”一章。

Servlet 引擎的事件可以分成 Servlet 环境事件和 Session 事件两种。通过 Servlet 引擎的事件机制可以做如下的工作：

- 管理数据库和各种资源的使用和释放。
- 建立计数机制。
- 监视 HTTP Session 的状态及属性。

Servlet 引擎的事件机制与 AWT 和 Swing 库中的事件模型是一样的事件委派模型（Delegation Event Model 或 DEM）。根据“观察者（Observer）模式”一章的讨论，DEM 是观察者模式的具体应用。

Servlet 环境的事件

当 Servlet 环境被创建、关闭，或者一个属性被加入到环境里、一个已经在环境里的

属性被删除或者修改时，都会产生 Servlet 环境事件。下面的表格给出了 Servlet 环境事件的类型，以及每一种事件所对应的接口和方法。如果一个对象要响应某一个事件，那么这个对象就应当实现对应的接口，以及事件所对应的方法，如下表所示。

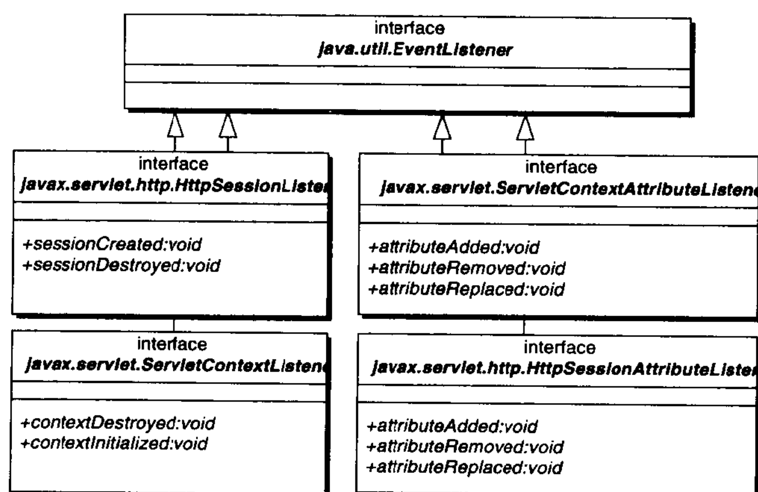
事件类型	接口	方法
Servlet 环境被创建	javax.Servlet.ServletContextListener	contextInitialized()
Servlet 环境就要关闭	javax.Servlet.ServletContextListener	contextDestroyed()
一个属性被加入	javax.Servlet.ServletContextAttributesListener	attributeAdded()
一个属性被删除	javax.Servlet.ServletContextAttributesListener	attributeRemoved()
一个属性被替换掉	javax.Servlet.ServletContextAttributesListener	attributeReplaced()

Session 的事件

当 Session 被激活（Activate）、钝化（Passivate），或者一个属性被加入到 Session 对象里、一个已经在 Session 里的属性被删除或者修改时，都会产生 Session 事件。下面的表各给出了 Session 事件的清单，每一事件所对应的时间监听器（listener）接口，以及所对应的方法，如下表所示。

事件类型	接口	方法
一个 session 被激活	javax.Servlet.http.HttpSessionListener	sessionCreated()
一个 session 被钝化（passivated）	javax.Servlet.http.HttpSessionListener	sessionDestroyed()
一个属性被加入	javax.Servlet.http.HttpSessionAttributesListener	attributeAdded()
一个属性被删除	javax.Servlet.http.HttpSessionAttributesListener	attributeRemoved()
一个属性被替换掉	javax.Servlet.http.HttpSessionAttributesListener	attributeReplaced()

上面的两种事件机制涉及到四个监听器接口，与 AWT 的事件接口一样，它们也都是扩展自 java.util.EventListener 的子接口。读者可以在其类图中清楚地看到每一个接口的情况，如下图所示。





怎样准备一个监听器（listener）类

要对某一个事件做出响应，就必须准备一个所谓的监听器（listener）类，这个类实际上扮演的就是观察者模式的角色。那么怎样才能准备一个监听器类呢？

（1）针对某一个事件的监听类必须实现上面所给出的对应于这个事件的接口。

（2）一个监听器类必须有一个公开的默认构造子（没有参量的构造子）。

（3）必须实现监听器接口所要求的方法。

关于具体的做法，可以参见下面的例子。

Servlet 环境监听器的例子

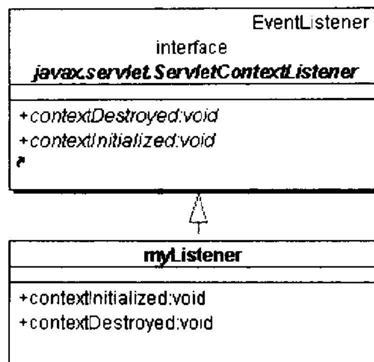
这里给出的是一个实现 ServletContextListener 接口的例子，如源代码清单 4 所示。ServletContextListener 接口要求实现两个方法，即 contextInitialized() 方法和 contextDestroyed()方法。

代码清单 4：一个 Servlet 监听器类的源代码

```
import javax.Servlet.*;
public final class myListener implements
    ServletContextListener
{
    /**
     *  当 J2EE 应用服务器
     *  （Application Server）
     *  的 Servlet Context 启动时，
     *  此方法会被触发。
     */
    public void contextInitialized(ServletContextEvent event)
    {
        //write your code here
    }

    /**
     *  当 J2EE 应用服务器
     *  的 Servlet Context 停掉时，
     *  此方法会被触发。
     */
    public void contextDestroyed(ServletContextEvent event)
    {
        //write your code here
    }
}
```

这个例子的类图如下图所示。



每当 J2EE 应用服务器（也就是 Application Server）的 Servlet Context 启动时，上面的第一事件方法会被触发；而每当 Servlet Context 停掉时，第二个事件方法会被触发。

Session 属性监听器的例子

Session 属性监听器负责监听 Session 属性的变化。加入一个新的属性，删除一个属性，或者修改一个属性都会导致一个 Session 属性监听器被触发。

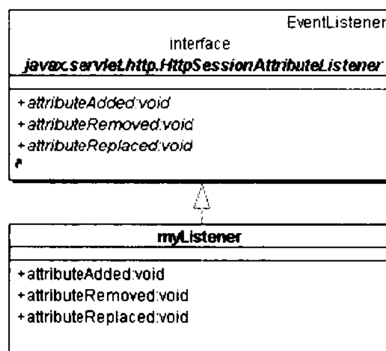
实现一个 Session 属性监听器类实际上非常简单，只是实现 HttpSessionAttributesListener 接口所要求的三个方法而已，如代码清单 5 所示。

代码清单 5：一个 Session 的监听器类的源代码

```
import javax.Servlet.*;
import javax.Servlet.http.HttpSessionBindingEvent;
import javax.Servlet.http.HttpSessionAttributeListener;
public final class myListener implements
    HttpSessionAttributesListener
{
    /**
     *     当 Session 中出现一个新的属性
     *     时，这个方法会被触发
     */
    public void attributeAdded(HttpSessionBindingEvent sbe)
    {
        // Write your code here
    }
    /**
     *     当 Session 中的一个属性
     *     被删除时，这个方法会被触发
     */
    public void attributeRemoved(HttpSessionBindingEvent sbe)
    {
        // Write your code here
    }
    /**
```

```
    *     当 Session 中的一个属性
    *     被换掉时，这个方法会被触发。
    */
   public void attributeReplaced(HttpSessionBindingEvent sbe)
   {
       // Write your code here
   }
}
```

这个例子的类图如下图所示。



命名规范

正如本书在“模版方法（Template Method）模式”一章中所指出的，使用恰当的命名规范给类和方法命名，可以帮助设计师之间有效地沟通。模版方法模式中的基本方法都是留给子类实现的，它们的名字应当以 do 开始。

HttpServlet 抽象类中提供了七个 do 方法，这些方法的命名都遵从这一命名规范。希望读者能理解到命名规范的重要性，并在自己的设计中遵守同样的规范。

参考文献

[SERVLET23] Sun Microsystem．Java Servlet 2.3 and JavaServer Pages 1.2 Specifications．2001

[BEA02] WebLogic 6.1 Documentation．BEA Systems．Application Events and Listeners，2000

[HALL00] Marty Hall．Core Servlets and Javaserver Pages．Prentice Hall，June 2000

第 38 章　观察者（Observer）模式

观察者模式是对象的行为模式[GOF95]，又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。

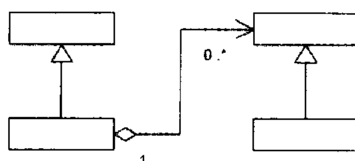
观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

38.1 引　　言

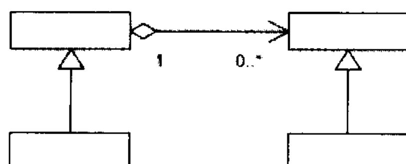
一个软件系统里面包含了各种对象，就像一片欣欣向荣的森林充满了各种生物一样。在一片森林中，各种生物彼此依赖和约束，形成一个个生物链。一种生物的状态变化会造成其他一些生物的相应行动，每一个生物都处于别的生物的互动之中。

同样，一个软件系统常常要求在某一个对象的状态发生变化的时候，某些其他的对象做出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合度的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计师需要使这些低耦合度的对象之间能够维持行动的协调一致，保证高度的协作（Collaboration）。观察者模式是满足这一要求的各种设计方案中最重要的一种。

根据观察者对象引用的存储地点，观察者模式的类图有微妙的区别。观察者模式的类图（一）如下图所示。



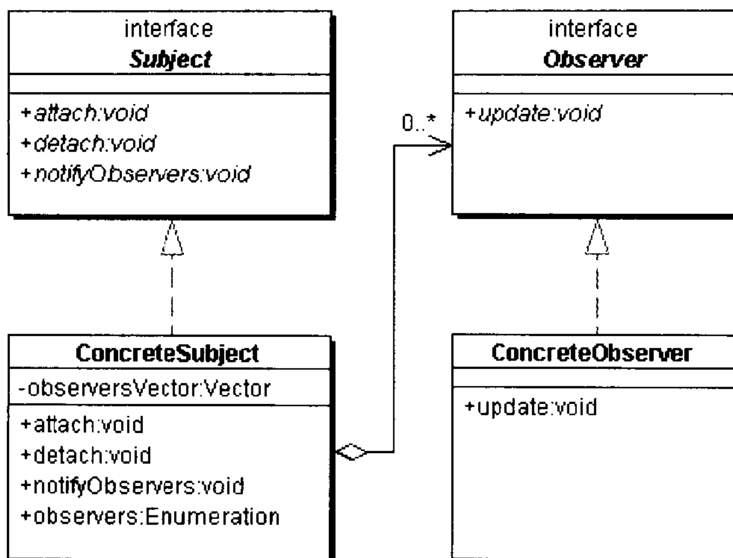
这种实现在传统的模式著作和讨论中比较常见，观察者模式的简略类图（二）如下图所示。



Java 语言提供的对观察者模式的支持属于此种结构。

38.2 观察者模式的结构

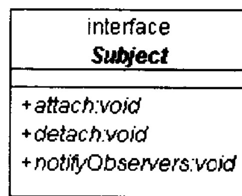
下面就以一个简单的示意性实现为例，讨论观察者模式的结构。这个实现的类图如下图所示。



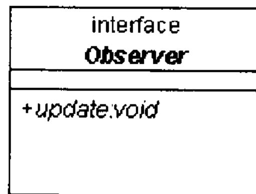
观察者模式的静态结构可以从类图中看清楚。

可以看出，在这个观察者模式的实现里有下面这些角色：

- 抽象主题（Subject）角色：主题角色把所有对观察者对象的引用保存在一个聚集（比如 Vector 对象）里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，主题角色又叫做抽象被观察者（Observable）角色，一般用一个抽象类或者一个接口实现。抽象主题角色如右图所示。

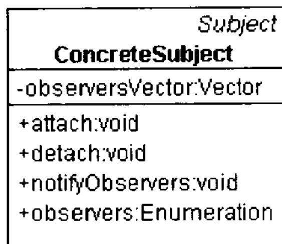


- 抽象观察者（Observer）角色：为所有的具体观察者定义一个接口，在得到主题的通知时更新自己。这个接口叫做更新接口。抽象观察者角色一般用一个抽象类或者一个接口实现，如右图所示。在这个示意性的实现中，更新接口只包含一个方法（即 update()方法），这个方法叫做更新方法。

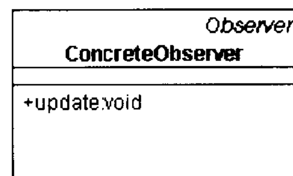


- 具体主题（ConcreteSubject）角色：将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色又叫作具体被观察者角色（Concrete Observable）。具体主题角色通常用一个具体子类实现，如下图所示。在这个示意性实现里，具体主题角色负责实现对观察者引用的聚集的管理方法。





- 具体观察者（ConcreteObserver）角色：存储与主题的状态自恰的状态。具体观察者角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。如果需要，具体观察者角色可以保存一个指向具体主题对象的引用。具体观察者角色通常用一个具体子类实现，如右图所示。



在这个示意性实现中，更新接口只有一个更新方法（也就是 update()方法）。

在类图中，从具体主题角色指向抽象观察者角色的合成关系，代表具体主题对象可以含有任意多个对抽象观察者对象的引用。当然，一个 Java 抽象类是不可能有实例的，因此这些引用的真实类型必然是 ConcreteObserver 类型；而这些引用的静态类型是 Observer 类型。这意味着主题对象不需要知道引用了哪些 ConcreteObserver 类型，而只知道抽象 Observer 类型。这就使得具体主题对象可以动态地维护一系列的对观察者对象的引用，并在需要的时候调用每一个观察者共有的 update()方法。

这种做法叫做“针对抽象编程”，对这一原则感兴趣的读者可以进一步阅读本书的“依赖倒转原则”一章。

下面给出一个示意性实现的 Java 代码。首先，Subject 接口实现抽象主题角色，如代码清单 1 所示。

代码清单 1：Subject 接口的源代码

```
public interface Subject
{
    /**
     * 调用这个方法登记一个新的观察者对象
     */
     public void attach(Observer observer);
    /**
     * 调用这个方法删除一个已经登记过的观察者对象
     */
     public void detach(Observer observer);
    /**
     * 调用这个方法通知所有登记过的观察者对象
     */
     void notifyObservers();
}
```

这个抽象主题接口规定出三个子类必须实现的操作，即 attach() 用来增加一个观察者对象；detach() 用来删除一个已经登记过的观察者对象；而 notifyObservers() 用来通知各个观察者更新它们自己。读者可以看出，attach()方法和 detach() 方法实际上是聚集的管理方法，抽象主题角色声明了这两个方法，实际上是要求具体子类维护一个以所有观察者对象为元素的聚集。

具体主题则是实现了抽象主题 Subject 接口的一个具体类，它给出了以上的三个操作的具体实现，如代码清单 2 所示。从下面的源代码可以看出，这里给出的 Java 实现使用了一个 Java 向量来保存所有的观察者对象，而 attach() 和 detach() 操作则是对此向量的元素增减操作。

在下一节给出的第二种实现方案中，这些局级管理方法被移到抽象主题角色中。

代码清单 2：ConcreteSubject 类的源代码

```
import java.util.Vector;
import java.util.Enumeration;
public class ConcreteSubject implements Subject
{
    private Vector observersVector =
        new java.util.Vector();
  /**
   * 调用这个方法登记一个新的观察者对象
   */
    public void attach(Observer observer)
    {
        observersVector.addElement(observer);
    }
  /**
   * 调用这个方法删除一个已经登记过的观察者对象
   */
    public void detach(Observer observer)
    {
        observersVector.removeElement(observer);
    }
  /**
   * 调用这个方法通知所有登记过的观察者对象
   */
    public void notifyObservers()
    {
        Enumeration enumeration = observers();
        while (enumeration.hasMoreElements())
        {
            ((Observer)
                enumeration.nextElement()).update();
        }
    }
  /**
```

```
     * 这个方法给出观察者聚集的 Enumeration 对象
     */
    public Enumeration observers()
    {
        return ((Vector)
            observersVector.clone()).elements();
    }
}
```



observers()方法给出的 Enumeration 对象是聚集的一个拷贝，从而使外界不能修改主题自己所使用的拷贝。

抽象观察者角色的实现实际上是最为简单的。它是只声明了一个方法（即 update()方法）的 Java 接口。这个方法被子类实现后，一被调用便更新观察者对象，如代码清单 3 所示。

代码清单 3：Observer 接口的源代码

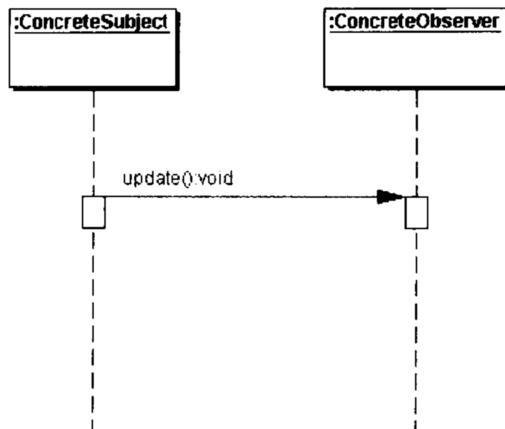
```
public interface Observer
{
    /**
     * 调用这个方法会更新自己
     */
    void update();
}
```

具体观察者角色的实现其实只涉及 update()方法的实现。这个方法怎么实现是与应用密切相关的，因此本书只给出一个框架，如代码清单 4 所示。读者可以在后面更加具体的例子中看到实现这个方法的细节。

代码清单 4：ConcreteObserver 类的源代码

```
public class ConcreteObserver implements Observer
{
    /**
     * 调用这个方法会更新自己
     */
    public void update()
    {
        System.out.println("I am notified");
    }
}
```

下图所示是具体主题角色调用具体观察者角色的更新方法时的时序图。

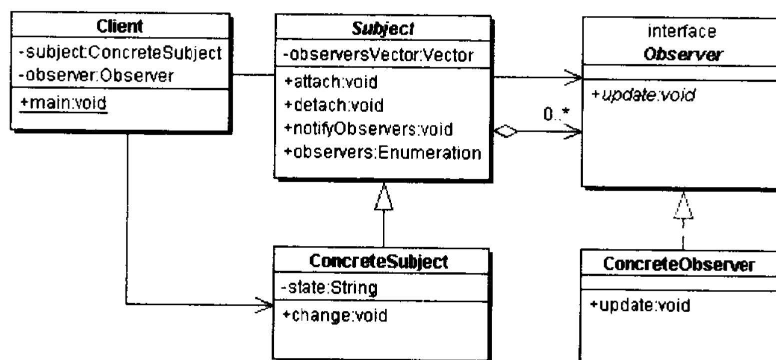


38.3 另一种实现方案

如果仔细考察主题对象的功能时，可以发现它必须使用一个 Java 聚集来维护一个对所有的观察者对象的引用。而在前面所给出的实现里面，管理这个聚集的方法是由抽象主题角色声明并由具体主题角色实现的。这才导致了类图中从具体主题角色到抽象观察者角色的连线。

但是，一个自然的问题就是，这些聚集管理方法难道在每一个具体主题角色中都不同吗？回答是否定的。换言之，在大多数情况下，这些聚集管理方法本身就是所有具体主题角色所共有的，因此这些方法连同聚集本身都可以移到抽象主题角色中去。同样，由于notifyObserver()方法依赖于聚集对象，也可以移到抽象主题角色中去。

这就是本节将要讨论的第二种实现方案。新实现的结构类图如下图所示。



可以看出，第二种实现与第一种实现的主要区别就是代表存储观察者对象的聚集连线是从抽象主题到抽象观察者。(为方便起见，本实现提供了一个客户端对象)

下面给出抽象主题的源代码，如代码清单 5 所示。显然，由于抽象主题角色必须实现一些方法，所以不能再使用 Java 接口，而应当采用一个抽象 Java 类。

代码清单 5：Subject 类的源代码

```
package com.javapatterns.observer.variation;
```



```
import java.util.Vector;
import java.util.Enumeration;
abstract public class Subject
{
    // 这个聚集保存了所有对观察者对象的引用
    private Vector observersVector =
        new java.util.Vector();
   /**
    * 调用这个方法登记一个新的观察者对象
    */
    public void attach(Observer observer)
    {
        observersVector.addElement(observer);
        System.out.println("Attached an observer.");
    }
   /**
    * 调用这个方法删除一个已经登记过的观察者对象
    */
    public void detach(Observer observer)
    {
        observersVector.removeElement(observer);
    }
   /**
    * 调用这个方法通知所有登记过的观察者对象
    */
    public void notifyObservers()
    {
        Enumeration enumeration = observers();
        while (enumeration.hasMoreElements())
        {
            ((Observer)
                enumeration.nextElement()).update();
        }
    }
   /**
    * 这个方法给出观察者聚集的 Enumeration 对象
    */
    public Enumeration observers()
    {
        return ((Vector)
            observersVector.clone()).elements();
    }
}
```

下面是具体主题角色的源代码，如代码清单 6 所示。显然，这个角色现在轻多了，只有一个会改变内部状态的商业方法。

代码清单 6：ConcreteSubject 类的源代码

```
package com.javapatterns.observer.variation;
public class ConcreteSubject extends Subject
{
    private String state;
    /**
     * 调用这个方法更改主题的状态
     */
    public void change(String newState)
    {
        state = newState;
        this.notifyObservers();
    }
}
```

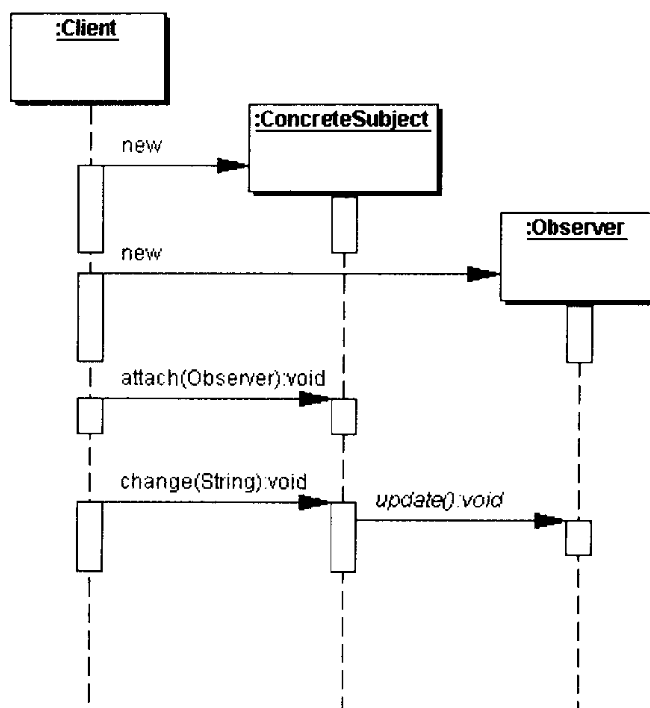
抽象观察者角色的源代码没有变化，为节省篇幅，不再列出。下面给出的是客户端的源代码，如代码清单 7 所示。

代码清单 7：Client 类的源代码

```
package com.javapatterns.observer.variation;
public class Client
{
    private static ConcreteSubject subject;
    private static Observer observer;
    public static void main(String[] args)
    {
        // 创建主题对象
        subject = new ConcreteSubject();
        // 创建观察者对象
        observer = new ConcreteObserver();
        // 将观察者对象登记到主题对象上
        subject.attach(observer);
        // 改变主题对象的状态
        subject.change("new state");
    }
}
```

在运行时，这个客户端首先创建了具体主题类的实例，以及一个观察对象。然后，它调用主题对象的 attach()方法，将这个观察者对象向主题对象登记，也就是将它加入到主题对象的聚集中去。

这时，客户端调用主题的商业方法，改变了主题对象的内部状态。主题对象在状态发生变化时，调用超类的 notifyObservers()方法，通知所有登记过的观察者对象。读者可以参考其系统时序图，如下图所示。



虽然观察者模式的实现方法可以由设计师自己确定，但是因为从 AWT 1.1 开始视窗系统的事件模型采用观察者模式，因此观察者模式在 Java 语言里的地位较为重要。正因为这个原因，Java 语言给出了它自己对观察者模式的支持。因此，本书建议读者在自己的系统中应用观察者模式时，不妨利用 Java 语言所提供的支持。

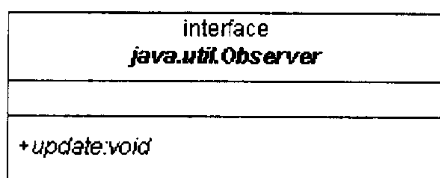
本节所给出的第二种实现方案与 Java 语言给出的对观察者模式的支持完全一致。

38.4　Java 语言提供的对观察者模式的支持

在 Java 语言的 java.util 库里面，提供了一个 Observable 类以及一个 Observer 接口，构成 Java 语言对观察者模式的支持。

Observer 接口

这个接口只定义了一个方法，即 update()方法。当被观察者对象的状态发生变化时，被观察者对象的 notifyObservers()方法就会调用这一方法。java.util 提供的 Observer 接口的类图如下图所示，其源代码如代码清单 8 所示。



代码清单 8：java.util.Observer 接口的源代码

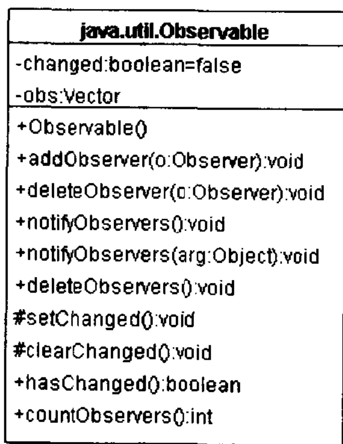
```
package java.util;
public interface Observer
{
    /**
     * 调用这个方法会更新自己
     */
    void update(Observable o, Object arg);
}
```

Observable 类

被观察者类都是 java.util.Observable 类的子类。java.util.Observable 提供公开的方法支持观察者对象，这些方法中有两个对 Observable 的子类非常重要：一个是 setChanged()，另一个是 notifyObservers()。第一个方法 setChanged()被调用之后会设置一个内部标记变量，代表被观察者对象的状态发生了变化。第二个是 notifyObservers()，这个方法被调用时，会调用所有登记过的观察者对象的 update()方法，使这些观察者对象可以更新自己。

java.util.Observable 类还有其他的一些重要的方法。比如，观察者对象可以调用 java.util.Observable 类的 addObserver()方法，将对象一个一个地加入到一个聚集上。当有变化时，这个聚集可以告诉 notifyObservers()方法哪些观察者对象需要通知。由于这个聚集是私有的，因此 java.util.Observable 的子对象并不知道哪些观察者对象一直在观察着它们。

Java 语言提供的被观察者的类图如下图所示。



被观察者类 Observable 的源代码如代码清单 9 所示。

代码清单 9：Observable 类的源代码

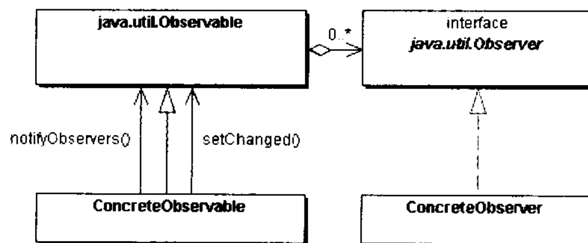
```
package java.util;
public class Observable
{
    private boolean changed = false;
```

```
private Vector obs;
/**
 *  构造子
 *  用 0 个观察者构造一个被观察者
 */
public Observable()
{
      obs = new Vector();
}
/**
 * 将一个观察者加到观察者聚集上面
 */
public synchronized void addObserver(Observer o)
{
      if (!obs.contains(o))
      {
            obs.addElement(o);
      }
}
/**
 * 将一个观察者对象从观察者聚集上删除
 */
public synchronized void deleteObserver(Observer o)
{
       obs.removeElement(o);
}
/**
 * 相当于 notifyObservers(null)
 */
public void notifyObservers()
{
notifyObservers(null);
}
/**
 * 如果本对象有变化（那时 hasChanged 方法会返回 true）
 * 调用本方法通知所有登记在案的观察者，即调用它们的 update()方法，
 * 传入 this 和 arg 作为参量
 */
public void notifyObservers(Object arg)
{
     /**
      * 临时存放当前的观察者的状态，参见备忘录模式
      */
     Object[] arrLocal;
     synchronized (this)
     {
```

```
            if (!changed)   return;
            arrLocal = obs.toArray();
            clearChanged();
        }
        for (int i = arrLocal.length-1; i=0; i--)
            ((Observer)arrLocal[i]).update(this, arg);
    }
    /**
     * 将观察者聚集清空
     */
    public synchronized void deleteObservers()
    {
    obs.removeAllElements();
    }
    /**
     * 将“已变化”设为 true
     */
    protected synchronized void setChanged()
    {
    changed = true;
    }
    /**
     * 将“已变化”重置为 false
     */
    protected synchronized void clearChanged()
    {
    changed = false;
    }
    /**
     * 探测本对象是否已变化
     */
    public synchronized boolean hasChanged()
    {
    return changed;
    }
    /**
     * 返回被观察对象（即此对象）的观察者总数
     */
    public synchronized int countObservers()
    {
        return obs.size();
    }
}
```

这个 Observable 类代表一个被观察者对象，有时称之为主题对象。一个被观察者对象可以有数个观察者对象，每个观察者都是实现 Observer 接口的对象。在被观察者对象发

生变化时，它会调用 Observable 的 notifyObservers 方法，此方法调用所有的具体观察者的 update()方法，从而使所有的观察者都被通知更新自己。使用 Java 语言提供的对观察者模式的支持的类图如下图所示。



必须指出的是，上面的这个实现与本章前面所谈到的观察者模式的第二种实现完全一致。在这个实现里，对所有的观察者对象的引用都存储在抽象主题角色里面，而管理聚集和通知登记过的观察者对象的方法也在抽象主题角色里面。

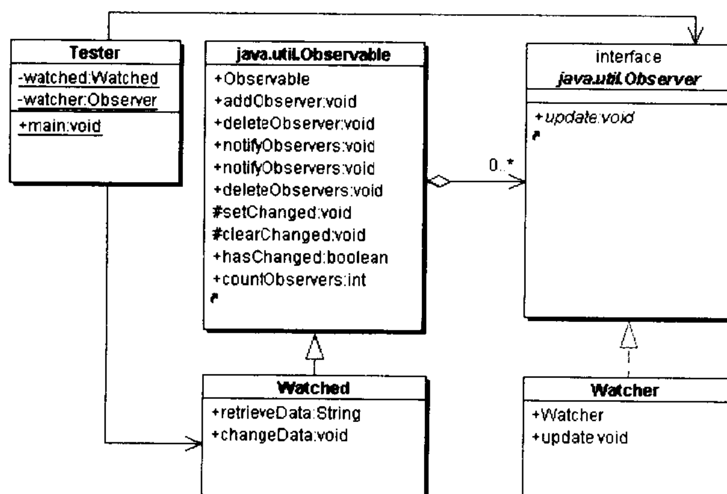
发通知的次序在这里没有指明。Observerable 类所提供的默认实现会按照 Observers 对象被登记的次序的相反次序通知它们，但是 Observerable 类的子类可以改掉这一次序。子类还可以在单独的线程里通知观察者对象，或者在一个公用的线程里按照次序执行。

最后值得注意的是，当一个可观察者对象刚刚被创建时，它的观察者集合是空的。两个观察者对象在它们的 equals()方法返回 true 时，被认为是两个相等的对象。

怎样使用 Java 对观察者模式的支持

为了说明怎样使用 Java 所提供的对观察者模式的支持，本节给出一个非常简单的例子。在这个例子里，被观察对象叫做 Watched，也就是被监视者；而观察者对象叫做 Watcher，也就是监视人的意思。Watched 对象继承自 java.util.Obsevable 类；而 Watcher 对象实现了 java.util.Observer 接口。另外有一个对象 Tester，扮演客户端的角色，其源代码如代码清单 10 所示。

这个系统简单的结构如下图所示。



在客户端改变 Watched 对象的内部状态时，Watched 就会通知 Watcher 采取必要的行动。

代码清单 10：Tester 类的源代码

```
package com.javapatterns.observer.watching;
import java.util.Observer;
public class Tester
{
    static private Watched watched;
    static private Observer watcher;
    public static void main(String[] args)
    {
        // 创建被观察者对象
        watched = new Watched();
        // 创建观察者对象，并将被观察者对象登记
        watcher = new Watcher(watched);
        // 给被观察者对象的状态赋值四次
        watched.changeData("In C, we create bugs.");
        watched.changeData("In Java, we inherit bugs.");
        watched.changeData("In Java, we inherit bugs.");
        watched.changeData("In Visual Basic, we visualize bugs.");
    }
}
```

在上面的客户端代码中，被观察者对象 watched 的状态被赋值四次，但状态只改变了三次。

下面是扮演被观察者角色的 Watched 类的源代码，这个类继承自 java.util.Observable 类，如代码清单 11 所示。

代码清单 11：Watched 类的源代码

```
package com.javapatterns.observer.watching;
import java.util.Observable;
public class Watched extends Observable
{
    private String data = "";
    /**
     *     取值方法
     */
    public String retrieveData()
    {
        return data;
    }
    /**
     *     改值方法
     */
    public void changeData(String data)
```



```
    {
        if ( !this.data.equals( data) )
        {
            this.data = data;
            setChanged();
        }
        notifyObservers();
    }
}
```

下面是扮演观察者角色的 Watcher 类的源代码，这个类实现了 java.util.Observer 接口，如代码清单 12 所示。

代码清单 12：Watcher 类的源代码

```
package com.javapatterns.observer.watching;
import java.util.Observable;
import java.util.Observer;
public class Watcher implements Observer
{
    /**
     *    构造子
     */
    public Watcher(Watched w)
    {
        w.addObserver(this);
    }
    /**
     *    更新方法
     */
    public void update( Observable ob, Object arg)
    {
        System.out.println("Data has been changed to:    '" + ((Watched)ob).retrieveData() + "'");
    }
}
```

可以看出，虽然客户端将 Watched 对象的内部状态赋值了四次，但是值的改变只有三次，如代码清单 13 所示。

代码清单 13：被观察者的内部状态发生了改变

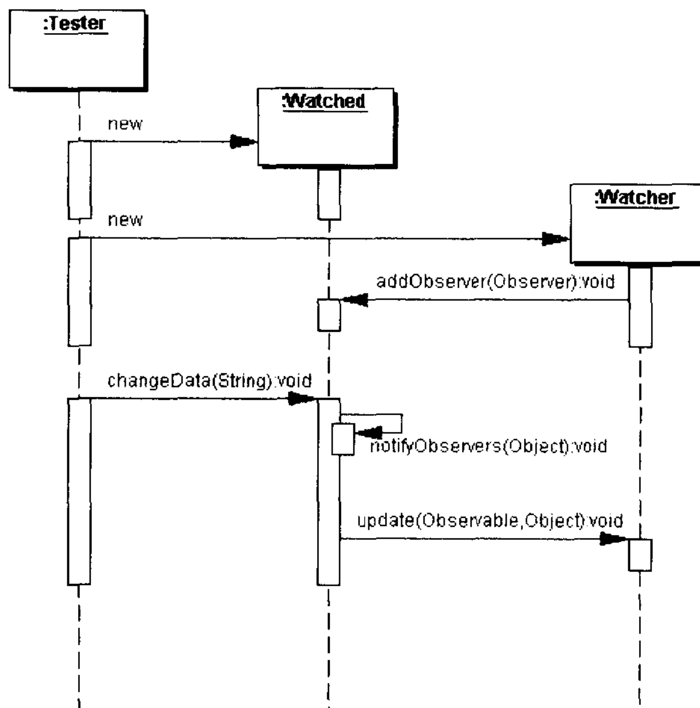
```
    watched.changeData("In C, we create bugs.");
    watched.changeData("In Java, we inherit bugs.");
    watched.changeData("In Java, we inherit bugs.");
    watched.changeData("In Visual Basic, we visualize bugs.");
```

相应地，Watcher 对象响应了这三次改变，运行程序，打印出的信息如代码清单 14 所示。

代码清单 14：运行的结果

```
Data has been changed to:   'In C, we create bugs.'
Data has been changed to:   'In Java, we inherit bugs.'
Data has been changed to:   'In Visual Basic, we visualize bugs.'
```

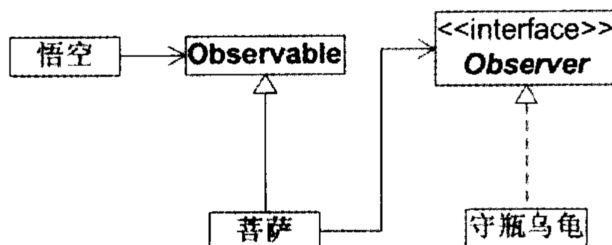
这个系统的活动时序图如下图所示。显然，Tester 对象首先创建了 Watched 和 Watcher 对象。在创建 Watcher 对象时，将 Watched 对象作为参量传入；然后 Tester 对象调用 Watched 对象的 changeData()方法，触发 Watched 对象的内部状态变化；Watched 对象进而通知实现登记过的 Watcher 对象，也就是调用它的 update()方法。



38.5 菩萨的守瓶龟

想当年齐天大圣为解救师傅唐僧，前往南海普陀山请菩萨降伏妖怪红孩儿："菩萨听说……恨了一声，将手中宝珠净瓶往海心里扑的一掼……只见那海当中，翻波跳浪，钻出个瓶来，原来是一个怪物驮着出来……要知此怪名和姓，兴风作浪恶乌龟。"

使用面向对象的语言描述，乌龟便是一个观察者对象，它观察的主题是菩萨。一旦菩萨将净瓶掼到海里，就象征着菩萨作为主题调用了 notifyObservers()方法。在西游记中，观察者对象有两个，一个是乌龟，另一个是悟空。悟空的反应在这里暂时不考虑，而乌龟的反应便是将瓶子驮回海岸。菩萨和菩萨的守瓶乌龟的类图如下图所示。



菩萨作为被观察者对象，继承自 Observable 类；而守瓶乌龟作为观察者，实现了 Observer 接口。这个模拟系统的实现可以采用 Java 对观察者模式的支持达成。

38.6 Java 中的 DEM 事件机制

AWT 中的 DEM 机制

责任链模式一章中曾谈到，AWT 1.0 的事件处理模型是基于责任链的。这种模型不适用于复杂的系统，因此，在 AWT 1.1 版本及以后的各个版本中，事件处理模型均为基于观察者模式的委派事件模型（Delegation Event Model 或 DEM）。

在 DEM 模型里面，主题（Subject）角色负责发布（publish）事件，而观察者角色向特定的主题订阅（subscribe）它所感兴趣的事件。当一个具体主题产生一个事件时，它就会通知所有感兴趣的订阅者。

使用这种发布-订阅机制的基本设计目标是提供一种将发布者与订阅者松散地耦合在一起的联系形式，以及一种能够动态地登记、取消向一个发布者的订阅请求的办法。显然，实现这一构思的技巧是设计抽象接口，并把抽象层和具体层分开。这在观察者模式里可以清楚地看到。

使用 DEM 的用语，发布者叫做事件源（event source），而订阅者叫做事件监听器（event listener）。在 Java 里面，事件由类代表，事件的发布是通过调用成员方法做到的。

关于 DEM 模型的细节请阅读本书的“专题：观察者模式与 SAX2 浏览器”一章。

Servlet 技术中的 DEM 机制

AWT 中所使用的 DEM 事件模型实际上被应用到了所有的 Java 事件机制上。Servlet 技术中的事件处理机制同样也是使用的 DEM 模型。关于这方面的讨论请见本书的“专题：Servlet 技术中的模式”一章。

SAX2 技术中的 DEM 机制

DEM 事件模型也被应用到了 SAX2 的事件处理机制上。关于这一点请见“观察者模式与 SAX2 浏览器”一章。

观察者模式的效果

观察者模式的效果有以下几个优点：

（1）观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者聚集，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

（2）观察者模式支持广播通信。被观察者会向所有的登记过的观察者发出通知。

观察者模式有下面的一些缺点：

（1）如果一个被观察者对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

（2）如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式时要特别注意这一点。

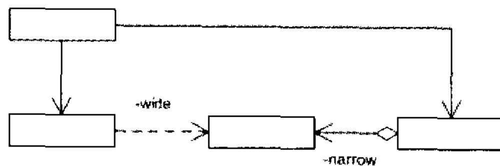
（3）如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

（4）虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。

38.7 观察者模式与其他模式的关系

观察者模式与备忘录模式的关系

观察者模式使用了备忘录模式（Memento Pattern），暂时将观察者对象存储在被观察者对象里面。备忘录模式的简略类图如下图所示。



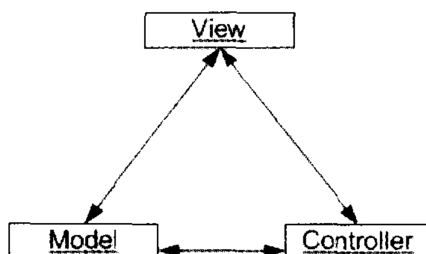
观察者模式与 MVC 模式的关系

MVC 属于架构模式，一个架构模式描述软件系统里的基本结构组织或纲要。在 MVC 模式里面有三个角色：模型（Model），视图（View）和控制器（Controller）。

观察者模式可以用来实现 MVC 模式。观察者模式中的主题角色便是 MVC 模式中的模型角色加控制器角色，而观察者角色便是 MVC 模型中的视图角色。MVC 模式的简略



类图如下图所示。



一般情况下，MVC 模式是观察者模式、合成模式、策略模式等设计模式的组合。读者可以从本书的“专题：MVC 模式与用户输入数据检查”一章读到关于 MVC 模式的专门讨论。

问答题

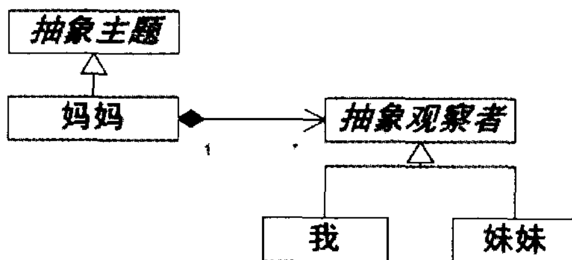
1. 我和妹妹跟妈妈说：“妈妈，我和妹妹在院子里玩。饭做好了叫我们一声。”请问这是什么模式？能否给出类图说明？

2. 请使用观察者模式解释 java.awt.image.ImageObserver 类的设计。

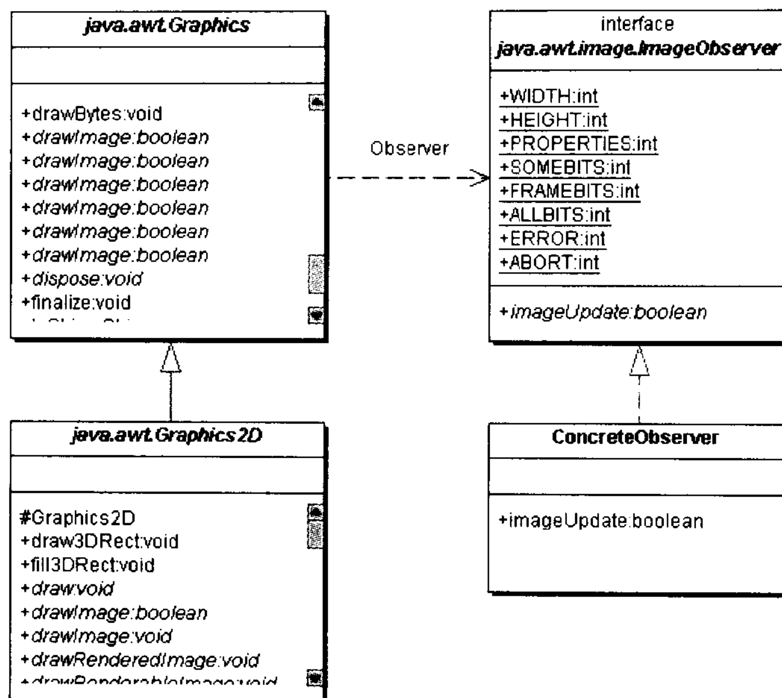
问答题答案

1. 这是观察者模式。“我”和“妹妹”让“妈妈”告诉我们饭做好了，这样我们就可以来吃饭了。换用较为技术化的语言来说，“我”和“妹妹”向系统的主题（“妈妈”）登记一个感兴趣的事件（饭熟了）。“妈妈”在事件发生时，通知系统的观察者对象（也就是“我”和“妹妹”），使其采取相应的行动（吃饭）。

系统的类图如下图所示。



2. ImageObserver 是观察者模式的应用。下图所示就是使用 ImageObserver 的示意图。



Graphics 类在调用 drawImage()方法时将一个观察者对象传入。当图像的状态有任何变化时，ImageObserver 对象就会被通知到。

ImageObserver 接口定义出一个非实时的更新接口（update interface），以便在图像的构造过程当中接收到图像的信息。

上面的 java.awt.Graphics2D 相当于一个具体的 Graphics 类，而 ConcreteObserver 对象则可以在收到图像信息时采取应该采取的行动。

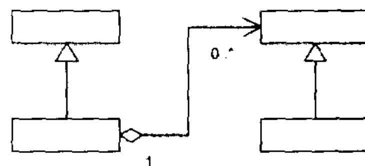
第 39 章　专题：观察者模式与 AWT 中的事件处理

在阅读本章之前，建议读者首先阅读本书的"观察者（Observer）模式"、"适配器（Adapter）模式"和"缺省适配（Default Adapter）模式"等章；在阅读过程当中，可以参考"责任链（Chain of Responsibility）模式"一章。

39.1　引　言

软件系统中的事件处理允许两个或者多个对象根据它们状态的变化进行通信和协调。在常见的事件处理模型中，对象通常被划分为三种：事件对象，事件制造者对象和事件接收者对象。一般而言，某一个对象是事件的制造者，其余对象是事件的接收者；而事件对象本身则封装了有关事件的信息。当事件制造者的内部状态发生变化时，会根据需要创建一个代表其状态变化的事件对象，并将它传给所有登记过的事件接收者对象。

Java 1.0 的事件处理机制是建立在责任链模式的基础上的，这样的事件处理机制不能满足大型应用系统的需要（对责任链模式感兴趣的读者请参见本书的"责任链模式"一章）。在 Java 1.1 版本里，Java 事件处理机制有了一个较大的变化：新的事件处理机制是建立在观察者模式基础之上的，以事件的委派为特征的委派事件模型（Delegation Event Model，DEM）。观察者模式的简略类图如右图所示。



Java 的 DEM 事件处理机制不仅被应用到了 AWT 库中，而且被应用到了 Swing 库中，以及一些其他厂家或组织提供的库，比如 SAX 库中。本章将集中讨论 AWT 中给出的 DEM 事件处理机制，以及它的结构和使用方法。

39.2　DEM 的结构

事件源对象

一个类要成为事件源并不需要实现任何接口或者继承任何类，但是一个事件源需要保持一个事件监听器的列表。调用 addXXXListener()方法增加一个监听器，调用 removeXXXListener()方法删除一个监听器，这两个方法都需要一个相应类型的监听器类

型。首先是 addXXXListner()方法的不同类型：

- addComponentListener()
- addFocusListener()
- addInputMethodListener()
- addKeyListener()
- addMouseListener()
- addMouseMotionListener()
- addPropertyChangeListener()

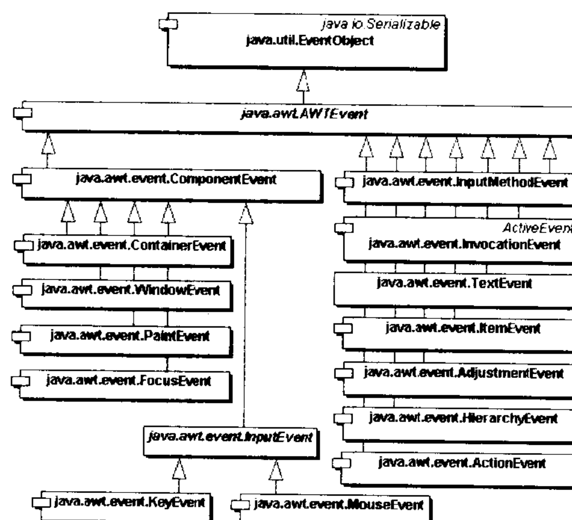
removeXXXListener()方法的不同类型如下：

- removeComponentListener()
- removeFocusListener()
- removeInputMethodListener()
- removeKeyListener()
- removeMouseListener()
- removeMouseMotionListener()
- removePropertyChangeListener()

所有的 AWT 构件都是 java.awt.Component 的子类，它们都从 Component 类继承了各个 addXXXListener()方法。

事件对象

在 DEM 中，每一种事件都有一个事件对象与之对应，而所有的 AWT 中的事件对象都是从 java.util.EventObject 继承而来的，每一种具体的事件对象都有一些额外的功能，java.awt 的各种事件类如下图所示。



由于旧事件模型的根节点叫做 Event，因此在新的 DEM 事件模型中，事件对象的树结构的根节点叫做 EventObject，以示区别。



在 AWT 库中，事件类 EventObject 只有一个直接的子类 AWTEvent，而后者是所有的 AWT 事件类的起源。

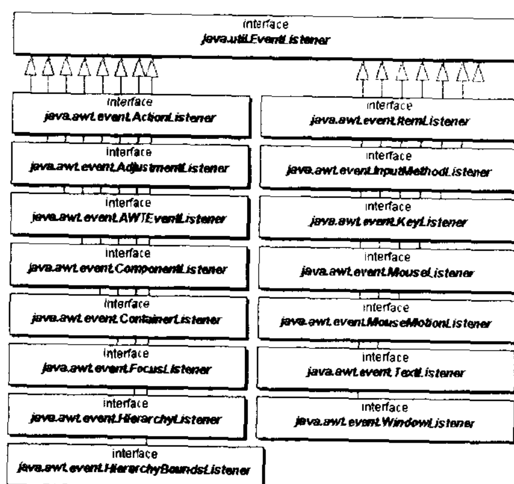
Swing 构件引进了更多的事件类，这些事件类不是从 AWTEvent 衍生来的，而是直接从 EventObject 衍生出来的。同样，读者可以设计自己的事件对象，直接继承自 EventObject 类。

事件对象封装了事件的源对象与事件监听器对象所需要的事件信息。有一些事件类，比如 PaintEvent，是不会被传递给监听器的，因此对 Java 程序员来说并没有什么用处。Java 应用软件程序设计师会接触到的是那些会被传递给监听器的事件类。这些事件类的名称如下：

- ActionEvent
- AdjustmentEvent
- ComponentEvent
- ContainerEvent
- FocusEvent
- ItemEvent
- KeyEvent
- MouseEvent
- TextEvent
- WindowEvent

事件监听器对象

事件监听器对象是当事件发生时被调用的对象。一个对象要成为事件监听器对象，必须实现事件监听器接口。AWT 库中所有的事件监听器接口都是 java.util.EventListener 接口的子接口，java.awt 的各种事件监听器接口的类图如下图所示。



应当指出的是，尽管 Swing 构件引入了很多的事件监听接口，但是 Swing 仍然广泛使用 AWT 所提供的各种事件接口进行事件处理。

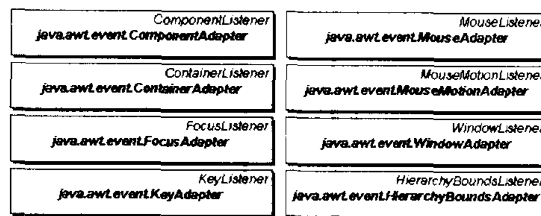
在 AWT 所给出的事件接口中，有以下接口声明了多于一个的方法，它们分别是：

- ComponentAdapter
- ContainerAdapter
- FocusAdapter
- KeyAdapter
- MouseAdapter
- MouseMotionAdapter
- WindowAdapter
- HierarchyBoundsAdapter

AWT 为这些接口提供了相应的适配器类，这些适配器类叫做事件监听适配器类。由于其他的几个接口仅仅声明了一个方法，因此给这些接口创建适配器类并没有任何益处。

事件监听适配器

下面的类图显示了八个事件监听适配器类，如下图所示。



这些适配器类为它们所实现的接口提供空的实现，这样一来，一个需要处理某个事件的应用类只需要继承相应的事件适配器类，并置换掉感兴趣的事件处理方法即可。相比之下，如果这个应用类实现相应的事件接口，则需要给出所有的事件处理方法，尽管大多数的事件处理方法是空的。显然前者使用起来更为方便。

读过缺省适配器模式的读者一定会发现，这些事件监听适配器都是缺省适配器模式的应用。关于适配器模式，请见本书的“适配器模式”一章。关于缺省适配器模式，请见本书的“缺省适配模式”一章。

尽管一下子引入了这么多的事件类、事件监听器接口、事件监听适配器类等，看上去好像很复杂，但实际上 AWT 的事件处理并不困难。只要读者理解了观察者模式，就会发现 AWT 的事件处理其实仅仅遵循了非常简单的原则。

语义事件与底层事件

AWT 区分了底层事件（Low-level Event）和语义事件（Semantic Event）。一个语义事件表达了用户所做的事情，比如使用鼠标单击一个按键就是一个语义事件。相应地，ActionEvent 就是一个语义事件。一个底层事件则是组成各种语义事件的单纯事件。比如鼠标单击事件由一个鼠标键按下事件和一个鼠标键释放事件组成。类似地，调整滚动条是一个语义事件，而拖曳鼠标则是一个底层事件。



下面是 AWT 中最重要的四个语义事件类：

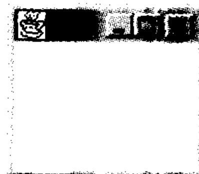
- ActionEvent（单击按键、选择菜单项、选择列表项、在文字框中输入文字等）
- AdjustmentEvent（调整滚动条）
- ItemEvent（用户在一组选择框中选择一个，或者在列表中选择一项）
- TextEvent（文字框的内容有变化）

下面是六个底层事件类：

- ComponentEvent（构件大小改变、位置变化、被隐藏、被显现）
- KeyEvent（一个键被按下或者释放）
- MouseEvent（鼠标被按下、释放、移动或者拖曳）
- FocusEvent（一个构件得到或失去聚焦）
- WindowEvent（视窗被激活、释放、最小化、关闭等）
- ContainerEvent（一个构件被增加到容器中或者从容器中删除）

本章在下面准备使用一个非常简单的鼠标事件的例子说明 AWT 事件与观察者模式的联系。

一个例子

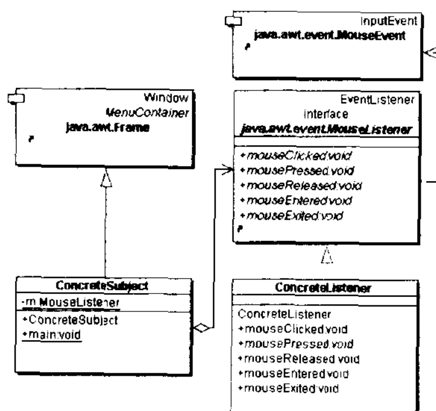


下面是一个非常简单的例子。显示一个空的视窗，接收鼠标单击事件。当用户在视窗上面单击鼠标时，系统需要捕获这个事件，并打印一个信息出来。系统运行时的类图如右图所示。

显然，所有的 Java 程序设计师都接受过比这更复杂的训练。本书选择这个例子，是为了说明其中的事件处理机制的应用以及所涉及的设计模式，特别给出几种不同的设计。为了说明 AWT 事件处理机制与观察者模式的关系，先来看一看下面的第一种设计。

第一种设计

在这个设计里面，事件的主题（事件源）和事件监听器是两个分开的对象，分别叫做 ConcreteSubject 和 ConcreteListener。前者是可见的视窗（Frame）对象，后者是在后台工作的事件监听对象。系统设计的类图如下图所示。



可以看出，Frame 实际上扮演了抽象主题角色，而 ConcreteSubject 则是具体主题角色；MouseListener 接口扮演抽象观察者角色，而 ConcreteListener 则是具体观察者角色。从主题角色传递给抽象观察者的事件对象就是 MouseEvent 对象，这也就是在上面的类图中 MouseListener 依赖于 MouseEvent 对象的原因。

上面的这个类图与观察者模式的类图是完全吻合的，非常形象地说明了 AWT 的鼠标事件处理机制就是观察者模式。

具体主题角色的源代码如代码清单 1 所示。

代码清单 1：ConcreteSubject 的源代码

```
package com.javapatterns.observerawt.mouse;
import java.awt.Frame;
import java.awt.event.MouseListener;
public class ConcreteSubject   extends Frame
{
    private static MouseListener m;
   /**
    *  缺省构造子
    */
    public ConcreteSubject()
    {
    }
    public static void main(String[] argv)
    {
        // 创建主题对象
        ConcreteSubject s = new ConcreteSubject();
        // 创建事件监听器对象
        m = new ConcreteListener();
        s.setBounds(100, 100, 100 , 100);
        // 将事件监听器对象登记到主题对象上
        s.addMouseListener(m);
        s.show();
    }
}
```

可以看出，这个具体主题角色其实自己并没有做什么，仅仅是继承了 Frame 的功能而已。而 Frame 的功能包括了维护一个监听器聚集，并提供各种管理监听器对象聚集的方法，比如 addMouseListener()等。

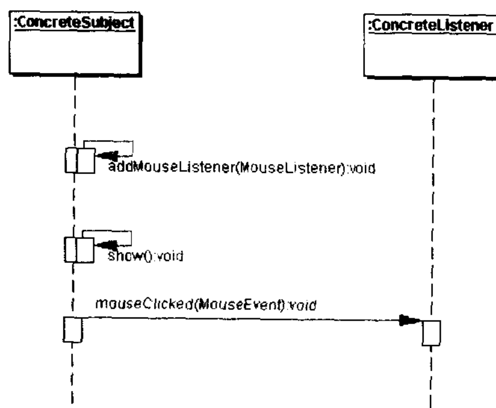
可以看出下面就是具体监听者类的源代码，如代码清单 2 所示，它实现了所有的 MouseListener 接口的全部五个事件方法。

代码清单 2：ConcreteListener 的源代码

```
package com.javapatterns.observerawt.mouse;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
public class ConcreteListener
```

```
    implements MouseListener
{
   /**
    *  缺省构造子
    */
    ConcreteListener()
    {
    }
   /**
    *  事件方法
    */
    public void mouseClicked(MouseEvent e)
    {
        System.out.println(e.getWhen());
    }
   /**
    *  事件方法(空实现)
    */
    public void mousePressed(MouseEvent e){ }
   /**
    *  事件方法(空实现)
    */
    public void mouseReleased(MouseEvent e){ }
   /**
    *  事件方法(空实现)
    */
    public void mouseEntered(MouseEvent e){ }
   /**
    *  事件方法(空实现)
    */
    public void mouseExited(MouseEvent e){ }
}
```

为了说明这个系统是如何运行的，最好看一看系统的时序图，如下图所示。



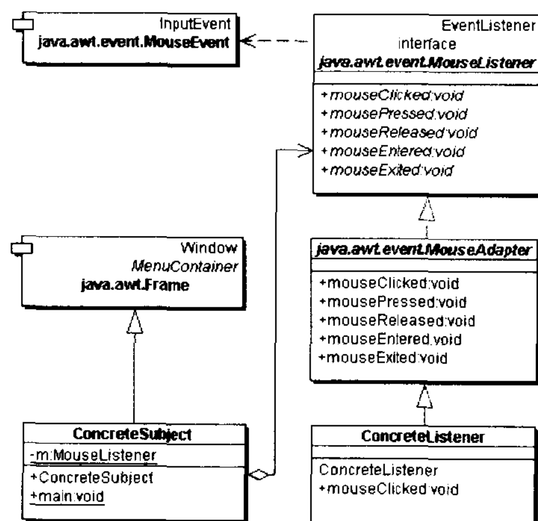
系统首先将 ConcreteListener 对象登记为事件处理器，然后调用 show()方法将 Frame

对象显示出来。这时如果用户单击鼠标按键的话，就会激发 mouseClicked()方法。

第二种设计

这种设计方案利用了事件适配器类 MouseAdapter，对第一种设计方案做了一点点修改。MouseAdapter 实现了 MouseListener 接口，为所有的方法都提供了空的实现。这样一来，ConcreteListener 类就只需要继承 MouseAdapter，并且置换掉 mouseClicked()方法即可，不需要为其他的方法提供空实现。

这一设计方案实际上就是第一种设计的变种，系统设计的类图如下图所示。



ConcreteSubject 类的源代码如代码清单 3 所示。

代码清单 3：系统的活动时序图

```
package com.javapatterns.observerawt.mouse3;
import java.awt.Frame;
import java.awt.event.MouseListener;
public class ConcreteSubject extends Frame
{
    private static MouseListener m;
   /**
    *  缺省构造子
    */
    public ConcreteSubject()
    {
    }
    public static void main(String[] argv)
    {
        // 创建主题对象
        ConcreteSubject s = new ConcreteSubject();
         // 创建事件监听器对象
      m = new ConcreteListener();
```

```
        s.setBounds(100, 100, 100 , 100);
        // 将事件监听器对象登记到主题对象上
        s.addMouseListener(m);
        s.show();
    }
}
```

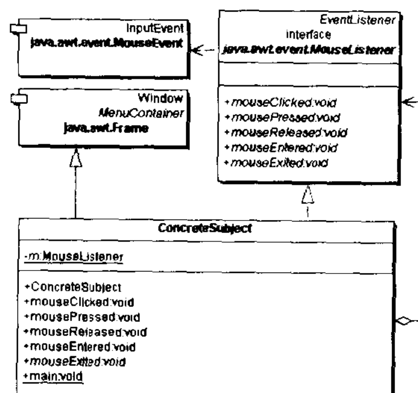
下面就是 ConcreteListener 类的源代码，如代码清单 4 所示。

代码清单 4：ConcreteListener 的源代码

```
package com.javapatterns.observerawt.mouse3;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
public class ConcreteListener
    extends MouseAdapter
{
   /**
    *  缺省构造子
    */
    ConcreteListener()
    {
    }
   /**
    *  鼠标事件方法
    */
    public void mouseClicked(MouseEvent e)
    {
        System.out.println(e.getWhen());
    }
}
```

第三种设计

在第三种设计中，事件的主题和监听器是合二为一的，叫做 ConcreteSubject。系统设计的类图如下图所示。



显然这个系统的中心就是一个 ConcreteSubject 类，这个类同时继承自 Frame 类，并且实现 MouseListener 接口。这个类从 Frame 类继承的功能包括维护一个监听器聚集，并提供各种管理监听器对象聚集的方法，比如 addMouseListener()等。同时它又实现了 MouseListener 接口，所以给出了所有 MouseListener 接口声明的方法。

下面就是这个类的源代码，如代码清单 5 所示。

代码清单 5：ConcreteSubject 的源代码

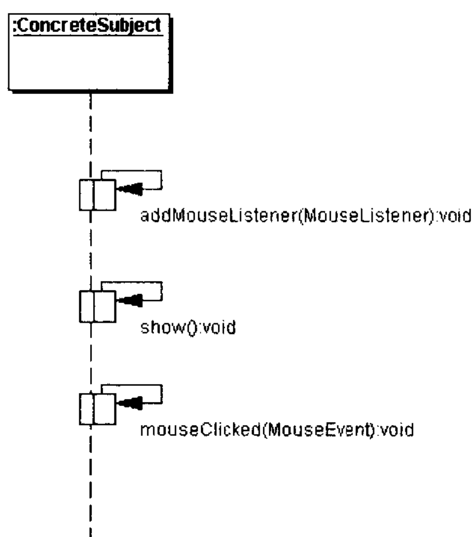
```
package com.javapatterns.observerawt.mouse1;
import java.awt.Frame;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
public class ConcreteSubject
    extends Frame implements MouseListener
{
   /**
    *  缺省构造子
    */
    public ConcreteSubject()
    {
    }
   /**
    *  事件方法
    */
    public void mouseClicked(MouseEvent e)
    {
        System.out.println(e.getWhen());
    }
   /**
    *  事件方法（空实现）
    */
    public void mousePressed(MouseEvent e){}
   /**
    *  事件方法（空实现）
    */
    public void mouseReleased(MouseEvent e){}
   /**
    *  事件方法（空实现）
    */
    public void mouseEntered(MouseEvent e){}
   /**
    *  事件方法（空实现）
    */
    public void mouseExited(MouseEvent e){}
    public static void main(String[] argv)
    {
        // 具体主题角色
```

```
        ConcreteSubject s = new ConcreteSubject();
        s.setBounds(100, 100, 100 , 100);

        // 设置事件监听器
        s.addMouseListener(s);

        s.show();
    }
}
```

为了说明这个系统是如何运行的，请看一看系统的时序图，如下图所示。



系统首先将自己登记为事件处理器，然后调用 show()方法将自己显示出来。这时如果用户单击鼠标按键的话，就会激发 mouseClicked()方法。

第四种设计

这第四种设计才是 Java 设计师的常用设计。在这个设计中，视窗是 ConcreteSubject 类，而事件监听器则由一个内部无名类实现，这个无名类实现了 MouseListener 接口并提供了所需的方法。在设计结构方面，请参考第一种设计的类图。

系统的源代码如代码清单 6 所示。

代码清单 6：ConcreteSubject 类的源代码

```
package com.javapatterns.observerawt.mouse2;
import java.awt.Frame;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
public class ConcreteSubject extends Frame
{
    /**
     *  缺省构造子
```

```
        */
        public ConcreteSubject()
        {
        }
        public static void main(String[] argv)
        {
            // 创建主题对象
            ConcreteSubject s = new ConcreteSubject();
            s.setBounds(100, 100, 100 , 100);
            // 声明一个无名事件监听器类
            // 将事件监听器对象登记到主题对象上
            s.addMouseListener( new MouseListener() {
              /**
               *  事件方法
               */
                public void mouseClicked(MouseEvent e)
                {
                    System.out.println(e.getWhen());
                }
              /**
               *  事件方法（空实现）
               */
               public void mousePressed(MouseEvent e){ }
              /**
               *  事件方法（空实现）
               */
                public void mouseReleased(MouseEvent e){ }
              /**
               *  事件方法（空实现）
               */
                public void mouseEntered(MouseEvent e){ }
              /**
               *  事件方法（空实现）
               */
                public void mouseExited(MouseEvent e){ }
            }
            );
            s.show();
        }
    }
```

AWT 的事件处理机制是建立在观察者模式的基础之上的，它为事件源和事件监听器解脱了耦合，并且使二者通过传递事件对象相互协调。

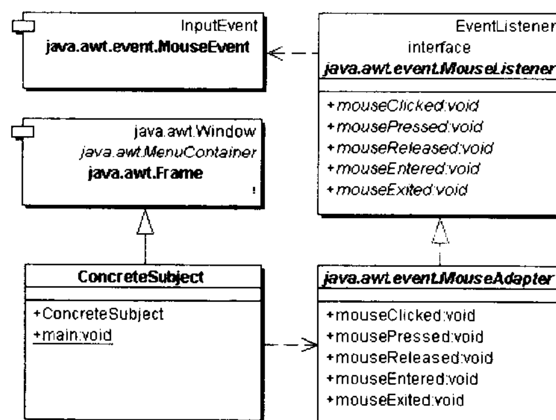
AWT 的事件处理还为使用者提供了事件适配器类，这是缺省适配器模式的应用。

问答题

请使用事件适配器类改写第四种设计方案。

问答题答案

修改后的设计方案如下图所示。



这个类图的缺点是不能体现出无名内部类。

系统只有一个类，就是 ConcreteSubject 类，其源代码如代码清单 7 所示。

代码清单 7：ConcreteSubject 类的源代码

```
package com.javapatterns.observerawt.mouse4;
import java.awt.Frame;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
public class ConcreteSubject extends Frame
{
    public ConcreteSubject()
    {
    }
    public static void main(String[] argv)
    {
        // 创建主题对象
        ConcreteSubject s = new ConcreteSubject();
        s.setBounds(100, 100, 100 , 100);
        // 声明一个无名事件监听器类
        // 将事件监听器对象登记到主题对象上
        s.addMouseListener( new MouseAdapter() {
          /**
           *  事件方法
           */
```

```
                public void mouseClicked(MouseEvent e)
                {
                    System.out.println(e.getWhen());
                }
            }
            );
            s.show();
        }
    }
```

参考文献

[COREJAVA] Cay S. Horstmann, Gary Cornell. Core Java 2 Volumn I – Fundamentals. The Sun Microsystem Press, 1999

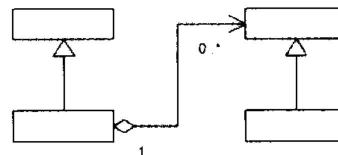
第 40 章　专题：观察者模式与 SAX2 浏览器

在阅读本章之前，请先阅读本书的“观察者（Observer）模式”、“专题：观察者模式与 AWT 中的事件处理”、“缺省适配（Default Adapter）模式”等章。关于 XML 和 SAX2 的基础知识，读者可以参考“专题：适配器模式与 XMLProperties”一章。

40.1 引　　言

观察者模式

从 AWT 1.1 版开始，Java 的事件处理机制就改成了 DEM（Delegation Event Model）机制，而这一机制是建立在观察者（Observer）模式的基础之上的。观察者模式的简略类图如右图所示。



SAX2 浏览器沿袭了 Java 的事件处理模式，也就是建立在观察者模式之上的 DEM 机制。与 AWT 和 Swing 一样，在 XML 文件的解析过程中，SAX2 使用观察者模式通知客户端一个 XML 文件含有什么内容。

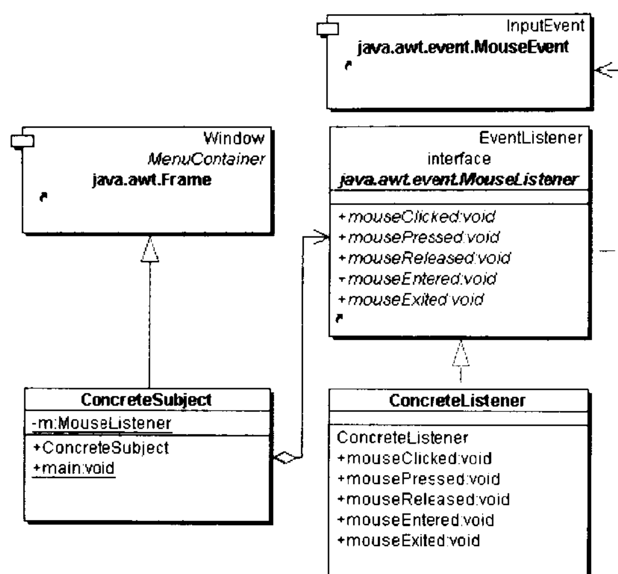
AWT 中的事件监听器接口

由于 AWT 的事件处理机制的关键就是 Java 程序师最熟悉的事件监听器接口，因此本章首先回顾一下 AWT 里的事件监听器接口是怎么工作的。

考虑一下前面“专题：观察者模式与 AWT 中的事件处理”一章所讨论的例子。

为了得到鼠标事件，一个客户类 ConcreteListener 根据需要实现了 MouseListener 接口。然后系统需要将这个 MouseListener 类型的对象登记到一个 Frame 构件（也就是 ConcreteSubject）上，以便可以接收到这个构件的鼠标事件。这样，用户如果在此构件上单击鼠标键的话，构件便会激发所有登记过的 MouseListener 对象，包括那个客户对象。

在前面的“专题：观察者模式与 AWT 中的事件处理”一章中给出的第一种设计可以参见下图所示的类图。



在这个例子里面，Frame 类扮演了主题角色，MouseListener 接口扮演了抽象观察者角色，而实现了 MouseListener 接口的客户类则扮演了具体观察者角色。

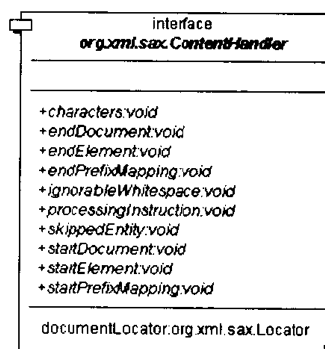
40.2 SAX2 是怎么工作的

SAX2 也是以类似的方式工作的。在 SAX2 里面，XMLReader 类扮演主体的角色，而 org.xml.sax.ContentHandler 接口扮演了观察者角色。AWT 与 SAX2 的最大区别在于 SAX2 并不允许向每一个 XMLReader 对象登记多于一个的 listener 对象，这是与通常的观察者模式的最大区别。

其次，SAX2 并不使用 java.util.EventObject 对象或者它的子类，因为一个 SAX2 事件的信息均是 XML 文件的标识结构，可以直接传递给事件监听器对象中的合适的事件方法。

40.3 ContentHandler 接口

几乎所有的 SAX2 应用程序都需要以某种形式实现 ContentHandler 接口。当处理一个特定的 XML 文件时，往往需要创建一个实现 ContentHandler 接口的类来处理特定的事件。因此，这个接口实际上就是 SAX 的 XML 文件处理的核心，ContentHandler 接口的类图如右图所示。



这个接口声明了一些方法，下面就来看看其中最重要的几个方法：

（1）void characters(char[] ch, int start, int length)：

这个方法用来处理在 XML 文件中读到的字符串。此方法的参数是一个 char 数组，以

及读到的这个字符串在这个数组中的起始位置和长度。有了这个方法，就可以很容易地用 String 类的一个构造方法将之转换成一个 String 类型：

```
new String(ch, start, length)
```

（2）void startDocument()：

当遇到 XML 文档开头的时候，上面这个方法就会被触发，程序可以利用这个机会做一些预先准备工作。

（3）void endDocument()：

和 XML 文档开头会触发的方法相对应，上面这个方法在文档结束的时候会被触发。用户程序可以利用这个机会做一些程序结束时需要做的工作。

（4）void startElement(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName, Attributes atts)

当读到一个开始标识符的时候，这个方法会被触发。在 SAX2 中提供了对名域的支持，这个参数中的 namespaceURI 就是域名，localName 是标识符名，qName 是标识符的修饰前缀，当没有使用名域的时候，这两个参数都为 null。而 atts 是这个标识符所包含的属性列表，通过 atts 可以得到所有的属性名和相应的属性值。

值得注意的是 SAX 的流处理方式，在遇到一个标识符的时候，它并不会记录下以前所碰到的标识符，也就是说，在 startElement()方法中，所有的信息就是标识符的名字和属性，至于标识符的嵌套结构、上层标识符的名字，是否有子元属等其他与结构相关的信息，都是不得而知的，都需要用户系统完成。

（5）void endElement(java.lang.String namespaceURI,
    java.lang.String localName,
    java.lang.String qName)

和上面的开始标识符所触发的方法相对应，在遇到结束标识符的时候这个方法会被触发。

下面给出这个接口的源代码，如代码清单 1 所示。

代码清单 1：ContentHandler 接口的源代码

```
package org.xml.sax;
public interface ContentHandler
{
    public void setDocumentLocator(Locator locator);
    public void startDocument() throws SAXException;
    public void endDocument() throws SAXException;
    public void startPrefixMapping(String prefix, String uri)
        throws SAXException;
    public void endPrefixMapping(String prefix)
      throws SAXException;
    public void startElement(String namespaceURI, String localName,
        String qualifiedName, Attributes atts) throws SAXException;
    public void endElement(String namespaceURI, String localName,
        String qualifiedName) throws SAXException;
    public void characters(char[] text, int start, int length)
```

```
        throws SAXException;
    public void ignorableWhitespace(char[] text, int start,
        int length) throws SAXException;
    public void processingInstruction(String target, String data)
        throws SAXException;
    public void skippedEntity(String name)
        throws SAXException;
}
```

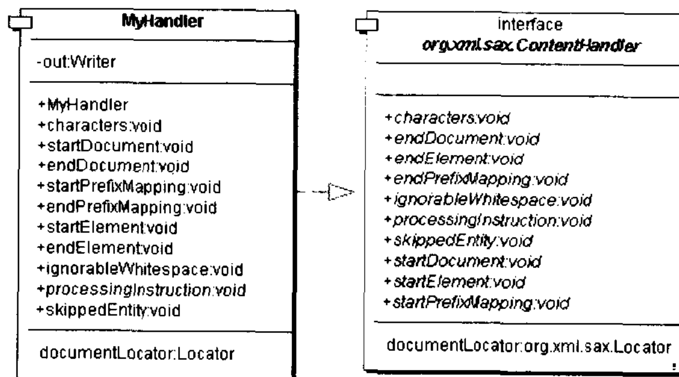
这个接口声明了 11 个方法，这 11 个方法并不一定全部用到，所以在使用的时候可能会有些不方便。为了解决这个问题，SAX 中还为其制定了一个辅助类：DefaultHandler，它实现了这个接口，但是其所有的方法体都为空。在实现的时候，你只需要继承这个类，然后置换掉相应的方法即可。

熟悉缺省适配模式的读者可以看出，DefaultHandler=就是省略适配模式的应用，本章在后面还会仔细讨论这一点。

40.4 怎样实现 ContentHandler 接口

为了能使读者将精力放到设计模式上面，本节特别选用一个非常简单的例子来说明怎样实现 ContentHandler 接口。

在这个浏览器系统里，程序首先读入一个 XML 文件的所有内容，然后去掉所有的标识符、注释、处理指令，打印出剩下的所有文字内容。本书准备将这个程序分成两个类，一个类实现 ContentHandler 接口，另一个类负责将文件导入到浏览器中。MyHandler 类的类图如下图所示。



下面就是第一个类 MyHandler 的源代码，如代码清单 2 所示。

代码清单 2：MyHandler 类实现了 ContentHandler 接口

```
import org.xml.sax.*;
import java.io.*;
public class MyHandler implements ContentHandler
{
```



```
    private Writer out;

    public MyHandler(Writer out)
    {
        this.out = out;
    }
    public void characters(char[] text, int start, int length)
       throws SAXException
    {
        try
        {
            out.write(text, start, length);
        }
        catch (IOException e)
        {
            throw new SAXException(e);
        }
    }
    // do-nothing methods
    public void setDocumentLocator(Locator locator) {}
    public void startDocument() {}
    public void endDocument() {}
    public void startPrefixMapping(String prefix, String uri) {}
    public void endPrefixMapping(String prefix) {}
    public void startElement(String namespaceURI, String localName,
       String qualifiedName, Attributes atts) {}
    public void endElement(String namespaceURI, String localName,
       String qualifiedName) {}
    public void ignorableWhitespace(char[] text, int start,
       int length) throws SAXException {}
    public void processingInstruction(String target, String data){}
    public void skippedEntity(String name) {}
}
```

上面的代码给出了 MyHandler 类，这个类实现了 ContentHandler 接口，因此它必须提供所有 11 个 ContentHandler 接口声明过的方法。由于这个简单的系统不需要所有的 11 个方法，因此这 11 个方法中有 10 个是空的。

除了这 11 个方法之外，MyHandler 类还有一个构造子，以及一个名为 out 的 Writer 类型的成员变量，而构造子给这个成员变量赋值。当然，只要需要，设计师可以给这个类增加其他不在 ContentHandler 接口所声明的 11 个方法中的新的方法。

可以看出，本系统所有的工作都是在 characters()方法中完成的。当浏览器读入标志符之间的内容时，它把所有的文字以字符数组 char[]的形式传给 characters()方法。所传入的文字的开始是由此方法的第二个参数 start 给出的，而字符的总数是由第三个参数 length 给出的。在这个例子里面，characters()方法所做的事情，就是将所传入的数组从 start 到 start+length 的元素全部传给 out 变量。

在本例子中，characters()方法调用 java.io.Writer 类的 write()方法，将数组的一些元素打印出来。由于 write()方法必须抛出一个 IOException，而 characters()并不抛出这个意外，因此在 characters()方法内部必须捕捉这个意外。此时，我们可以简单地将这个意外打印到控制台，但是这里选择将这个 IOException 对象封装在一个 SAXException 对象里，并重新抛出去。客户端可以捕捉这个 SAXException 对象，并意识到系统出现了问题。如果客户端需要知道具体的意外信息，客户端可以调用 getException()方法做到这一点。

40.5 怎样使用 ContentHandler

由于 MyHandler 并没有任何的代码读入一个文件，因此 MyHandler 自己并不能完成浏览任务。下面给出一个 MyReader 类，这个类负责读入文件和调用 MyHandler 对象，如代码清单 3 所示。

代码清单 3：MyReader 类的源代码

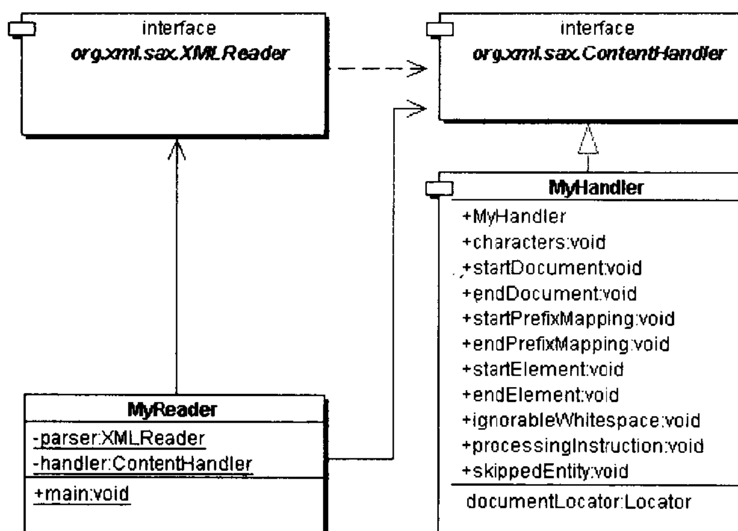
```
import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;
import java.io.*;
public class MyReader
{
    public static void main(String[] args)
    {
        if (args.length <= 0)
        {
            System.out.println( "Usage: java ExtractorDriver url" );
            return;
        }
        try
        {
            XMLReader parser = XMLReaderFactory.createXMLReader();
            Writer out = new OutputStreamWriter(System.out);
            ContentHandler handler = new TextExtractor(out);
            parser.setContentHandler(handler);
            parser.parse(args[0]);

            out.flush();
        }
        catch (Exception e)
        {
            System.err.println(e);
        }
    }
}
```

在使用这个程序时，需要修改 JVM 的系统性质。下面是一个示范命令：

```
java -Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser MyReader myxml.xml
```

为了显示观察者模式在这里的应用，下面特地将 MyHandler 类的类图完整给出，如下图所示。



在这里 MyReader 扮演主题角色，注意 MyReader 并没有实现 XMLReader 接口，而是引用一个 XMLReader 类型的对象 parser，这个对象是由 XMLReaderFactory 生成的。MyReader 将自己所持有的对这个对象的引用传递给 parser 对象。因此，parser 是真正的主题对象，通过 MyReader 的传递，它持有一个 ContentHandler 类型的观察者对象。

显然，MyHandler 是具体观察者者，而 ContentHandler 则是抽象观察者，前者实现了后者所声明的所有 11 个接口。

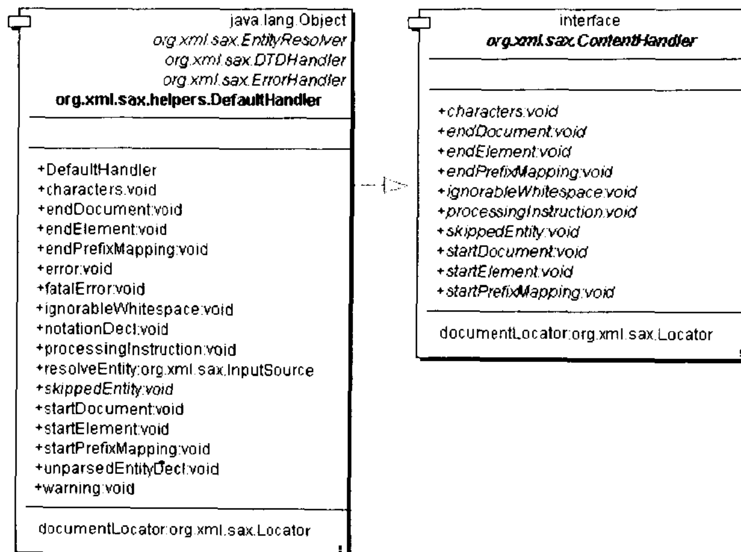
显然，这是一个观察者模式的应用。

40.6 缺省适配模式与 DefaultHandler

MyHanlder 类只使用了 ContentHandler 接口所要求的 11 个方法中的一个，所有其他 10 个方法都是空的。实际上，很少有 SAX2 程序会需要实现全部的 11 个方法。因此，熟悉缺省适配模式的读者会想到，在这里使用缺省适配模式是一个好主意。

关于缺省适配模式，请见本书的“缺省适配（Default Adapter）模式”一章。

事实上，SAX2 确实提供了这样一个缺省适配类，即 org.xml.sax.helpers.DefaultHandler 类。缺省适配模式的应用类图如下图所示。



下面就是这个缺省适配类的源代码，如代码清单 4 所示。

代码清单 4：DefaultHandler 类的源代码

```
package org.xml.sax.helpers;
import org.xml.sax.InputSource;
import org.xml.sax.Locator;
import org.xml.sax.Attributes;
import org.xml.sax.EntityResolver;
import org.xml.sax.DTDHandler;
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
public class DefaultHandler
    implements EntityResolver, DTDHandler, ContentHandler, ErrorHandler
{
    ////////////////////////////////////
    // Default implementation of
    // the EntityResolver interface.
    ////////////////////////////////////
    public InputSource resolveEntity (String publicId,
        String systemId)
    throws SAXException
    {
    return null;
    }
    ////////////////////////////////////
    // Default implementation of DTDHandler interface.
    ////////////////////////////////////
    public void notationDecl (String name, String publicId,
```



```
    String systemId)
throws SAXException
{
// no op
}
public void unparsedEntityDecl (String name, String publicId,
                String systemId, String notationName)
throws SAXException
{
// no op
}
////////////////////////////////////
// Default implementation of ContentHandler interface.
////////////////////////////////////
public void setDocumentLocator (Locator locator)
{
// no op
}
public void startDocument ()
throws SAXException
{
// no op
}
public void endDocument ()
throws SAXException
{
// no op
}
public void startPrefixMapping (String prefix, String uri)
throws SAXException
{
// no op
}
public void endPrefixMapping (String prefix)
throws SAXException
{
// no op
}
public void startElement (String uri, String localName,
              String qName, Attributes attributes)
throws SAXException
{
// no op
}
public void endElement (String uri, String localName,
    String qName)
```



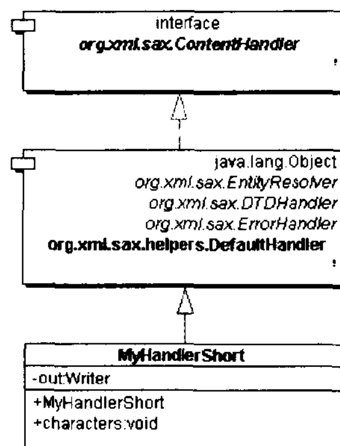

```
throws SAXException
{
// no op
}
public void characters (char ch[], int start, int length)
throws SAXException
{
// no op
}
public void ignorableWhitespace (char ch[], int start, int length)
throws SAXException
{
// no op
}
public void processingInstruction (String target, String data)
throws SAXException
{
// no op
}
public void skippedEntity (String name)
throws SAXException
{
// no op
}
////////////////////////////////////////
// Default implementation of the ErrorHandler interface.
////////////////////////////////////////
public void warning (SAXParseException e)
throws SAXException
{
// no op
}
public void error (SAXParseException e)
throws SAXException
{
// no op
}

public void fatalError (SAXParseException e)
throws SAXException
{
throw e;
}
}
```

设计师可以从 DefaultHandler 类继承，并置换掉系统感兴趣的方法，而不需要提供所

有的 11 种方法的实现。

这样，如果重新使用扩展 DefaultHandler，MyHandler 类可以更加简洁。系统的类图如下图所示。



新版的 MyHandler 类改名为 MyHandlerShort，以示区别，它是建立在扩展缺省适配类 DefaultHandler 基础之上的新版本 MyHandlerShort，其源代码如代码清单 5 所示。

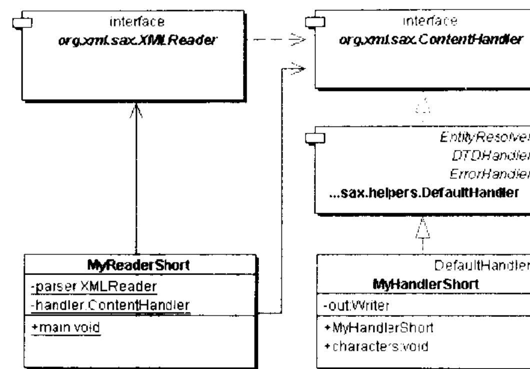
代码清单 5：MyHandlerShort 的源代码

```
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import java.io.*;
public class MyHandlerShort extends DefaultHandler
{
    private Writer out;

    public MyHandlerShort(Writer out)
    {
        this.out = out;
    }

    public void characters(char[] text, int start, int length)
     throws SAXException
     {
        try
        {
            out.write(text, start, length);
        }
        catch (IOException e)
        {
            throw new SAXException(e);
        }
    }
}
```

为了显示观察者模式在这里的应用，下面特地将系统的类图完整给出，如下图所示。



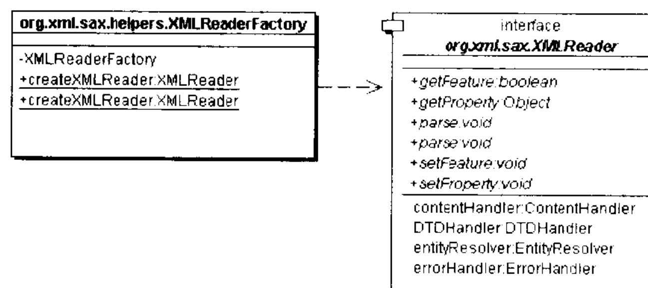
在这个系统里，MyReaderShort 通过 XMLReaderFactory 创建一个 XMLReader 类型的对象 parser。MyReaderShort 将自己所持有的对这个对象的引用传递给 parser 对象。parser 是真正的主题对象，通过 MyReader 的传递，它持有一个 ContentHandler 类型的观察者对象。

显然，MyHandlerShort 是具体观察者，而 ContentHandler 则是抽象观察者，前者实现了后者所声明的 11 个方法中的一个。

显然，这也是一个观察者模式的应用。

40.7 简单工厂模式的应用

XMLReaderFactory 使用了简单工厂模式，动态地创建类型为 XMLReader 的对象。XMLReaderFactory 创建 XMLReader 类型的对象的结构图，如下图所示。



XMLReaderFactory 提供了两种不同的静态方法，适用于不同的驱动类参数。

参考文献

[HAROLD02] Elliotte Rusty Harold. Processing XML with Java. Addison-Wesley，2002

第 41 章　专题：观察者模式与 Swing 定时器

在阅读本章之前，请先阅读本书的“观察者（Observer）模式”、“专题：观察者模式与 AWT 中的事件处理”等章。

41.1　为什么需要定时器

显然，在很多系统中，定时器都扮演着重要的角色。有些任务必须按照一定的时间安排反复执行，有些任务必须在一段时间间隔后一次性执行，这些情况都需要运用定时器。

很多操作系统都提供定时功能，比如 WIN 32 提供 SetTimer 这样的 API 函数来设定一个定时任务，同时提供 AT 命令集；UNIX 则提供 SIGALRM 信号和 cronjob 完成类似的工作。大多数编程语言也都提供定时功能，比如 Visual Basic 就提供一个 Timer 控件，让程序员在其 Timer 事件程序里面调用需要定时执行的操作。

从 Java 1.3 版本开始，定时器成为 java.util 库包的成员。本书“定时器与击鼓传花”一章已经对 java.util.Timer 的使用做了详尽的解释。值得研究的还有一个同时出现在 Swing 库包中的定时器类，即 javax.swing.Timer 类。本章将详细说明这个 Swing 库包中的定时器的使用，它所蕴涵的模式以及它与 java.util.Timer 的区别。

首先，为了在名称上区分两种不同的定时器，本章根据它们的来源把 java.util.Timer 叫做 Utility 定时器，而把 Swing 库包所提供的 Timer 叫做 Swing 定时器。

41.2　Utility 定时器与 Swing 定时器的区别

显然，这两种定时器所提供的功能在大体上是类似的。Utility 定时器是一个为一般用途设计的定时器，而 Swing 定时器是专为 Swing 库包准备的。Swing 定时器在一个线程里面为任意多的操作定时，这个定时器往往用在变换文字框里的光标、定时显示和隐藏 tooltips（显示提示的标签控件）等用户界面上。

Swing 的线程安全性

如同 AWT 一样，Swing 不是线程安全的，它们均使用单一的线程处理所有来自操作系统的事件。当接到来自操作系统的事件时，会将事件按照队列方式排序，然后依次判断

事件的类型，再通知有关的 listener 对象。比如在 Windows 操作系统中接到一个 wm_mouseMove 消息的话，Swing 就会向所有已登记的 MouseMotionListener 类型的对象发出 mouseMoved()消息。

像 mouseMoved()这样的事件处理器与依次处理 OS 层次的事件队列的操作是在同一个线程里面的。当事件处理操作运行时，GUI 等于被锁住了。这时，如果用户按某个按键的话，OS 会照常传入事件消息，只是因为事件处理线程被占线，因此事件处理机制不会立即处理这时候传入的消息。因此为了使系统的 GUI 总是保持对用户事件的响应，事件处理操作就应当简短和快速，所有耗费时间的操作都应当利用新的线程处理。

Swing 的类没有一个是线程安全的。如果 Swing 类都设计成线程安全的，引发的复杂性和运行开销之大很难被接受。因此，Swing 的类根本就不是设计用来同时处理多线程的。缺乏线程安全性意味着一旦系统通过调用诸如 setVisible()、pack()或其他的方法，将一个视窗变成可见的时，事件处理线程便开始运行，这时就无法安全地从 Swing 自己的事件处理器线程之外的任何线程里调用这个视窗的方法。

这通常不是一个问题，因为 Swing 构件的方法通常被 Swing 的事件处理器自己调用，注意所有的 listener 方法都是在这个线程上运行的。例如，一个视窗的事件处理器 mouseMoved()可能会决定向某个构件发出 repaint()消息，这种调用是安全的，因为 mouseMoved()方法是在 Swing 自己的线程上运行的。

一般来说，如果所需要定时的功能与 Swing 无关的话，应当使用 Utility 定时器。而如果设计的系统和所定时的功能是与 Swing 密切相关的话，就应当考虑使用 Swing 定时器。

两种定时器在性能上的区别

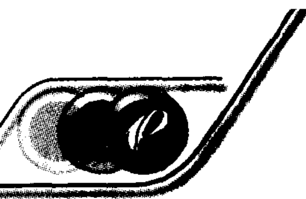
这两种定时器在性能上最主要的区别有如下两点：

（1）Swing 定时器类使用一个线程处理所有的定时器对象，因而不宜用来同时处理大量的定时器对象；而 Utility 定时器则不然，它为每一个定时器对象提供一个独立的线程，因此如果所设计的系统要求大量的定时器对象的话，则使用 Utility 定时器才可满足需要。

（2）Swing 定时器类的所有定时器对象所用的线程与事件处理器使用的是同一个线程，因此适合于在 Swing 构件中使用。Utility 定时器一般不适用于处理 Swing 构件的定时工作。

两种定时器在使用上的区别

这两种定时器在使用上的最主要的区别在于使用 Swing 定时器类时需要创建一个 Action 对象，以封装被定时的行为；而 Utility 定时器则需要为每一个被定时的任务创建一个 TimerTask 对象，以封装被定时的行为。比较之下，前者很接近观察者模式。



41.3　使用 Swing 定时器的方法

使用 Swing 定时器类时，必须注意到的是，Swing 定时器类使用一个（私有的）线程处理所有的定时器实例。它为需要定时的操作设定调用时间，然后使自己的线程休眠（sleep）。这种设计方式使 Swing 定时器类不能处理大量的定时器。

Swing 定时器对象会在设定好的时间触发一个事件。要实现 Swing 定时器，必须向计时器提供一个 Action 对象，并实现 Action 对象的 actionPerformed 方法，以便执行所指定的操作。下图所示的小程序会每隔 0.3 秒钟变换一次光标，其源代码如代码清单 1 所示。

Applet **RotatingCursorCompact**
-updateCursorAction:Action=new AbstractActi
+drawCursor(cursorType:int):void +init():void

代码清单 1：使光标旋转的源代码

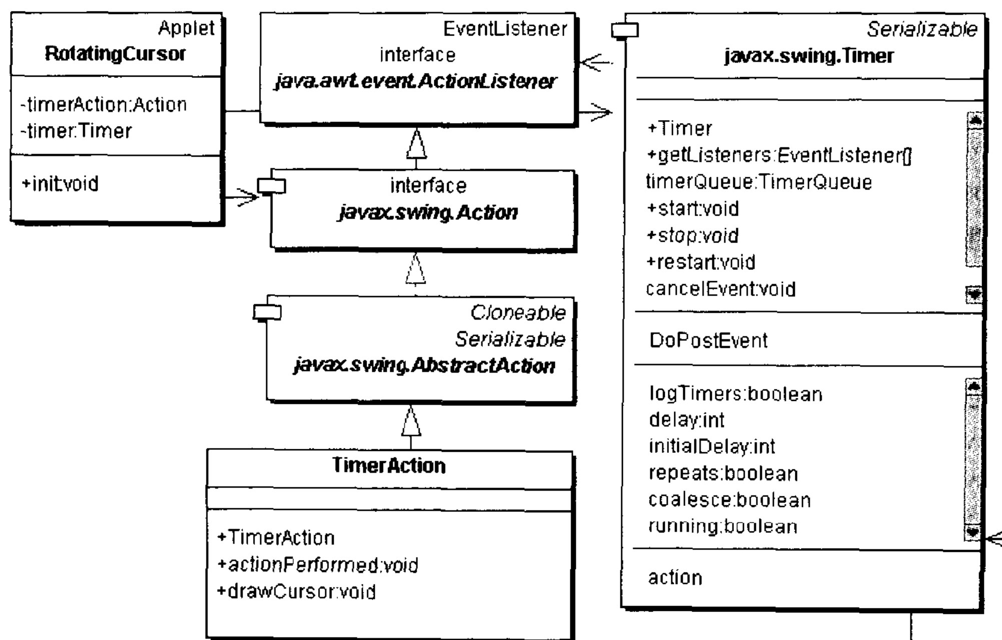
```
package com.javapatterns.observertimer.cursor;
import javax.swing.Action;
import javax.swing.AbstractAction;
import javax.swing.Timer;
import java.awt.Cursor;
import java.awt.event.ActionEvent;
import java.applet.Applet;
public class RotatingCursorCompact extends Applet
{
    private Action updateCursorAction = new AbstractAction()
    {
        int ind = 0;
        public void actionPerformed(ActionEvent e)
        {
            if (ind == 0)
            {
                drawCursor(Cursor.NE_RESIZE_CURSOR);
            }
            else if (ind == 1)
            {
                drawCursor(Cursor.N_RESIZE_CURSOR);
            }
            else if (ind == 2)
            {
                drawCursor(Cursor.NW_RESIZE_CURSOR);
            }
```

```
            else if (ind == 3)
            {
                drawCursor(Cursor.W_RESIZE_CURSOR);
            }
            else if (ind == 4)
            {
                drawCursor(Cursor.SW_RESIZE_CURSOR);
            }
            else if (ind == 5)
            {
                drawCursor(Cursor.S_RESIZE_CURSOR);
            }
            else if (ind == 6)
            {
                drawCursor(Cursor.SE_RESIZE_CURSOR);
            }
            else if (ind == 7)
            {
                drawCursor(Cursor.E_RESIZE_CURSOR);
                ind = -1;
            }
            ind++;
        }
    };
    public void drawCursor(int cursorType)
    {
        setCursor(Cursor.getPredefinedCursor(cursorType));
    }
    public void init()
    {
        new Timer(300, updateCursorAction).start();
    }
}
```

41.4 观察者模式的应用

上一节的这个程序的好处是比较紧凑，不好之处是不容易看清楚所设计的角色之间的关系。因此，下面提供了一个不同的版本。

首先给出这个版本所涉及的角色类图，如下图所示。



从这个类图可以看出，这个版本实际上是观察者模式的应用，其中涉及的角色有如下几个：

- （具体）被观察者角色：由定时器 Timer 类扮演。
- 具体观察者角色：由 TimerAction 类扮演。
- 抽象具体观察角色：由 ActionListener 类扮演。

下面给出的是客户端 RotatingCursor 的源代码，如代码清单 2 所示。可以看出，客户端首先创建一个定时器的观察者对象，即 TimerAction 对象，然后创建一个定时器。在创建定时器时，将一个 Action 类型的对象（即 TimerAction 对象）作为观察者对象传给定时器的构造子。

代码清单 2：RotatingCursor 类的源代码

```
package com.javapatterns.observertimer.cursor;
import javax.swing.Action;
import javax.swing.AbstractAction;
import javax.swing.Timer;
import java.awt.Cursor;
import java.awt.event.ActionEvent;
import java.applet.Applet;
public class RotatingCursor extends Applet
{
    private Action timerAction;
    private Timer timer;
    public void init()
    {
        timerAction = new TimerAction(this);
        timer = new Timer(300, timerAction);
```

```
            timer.start();
        }
    }
```

下面就是观察者 TimerAction 类的源代码，如代码清单 3 所示。RotatingCursor 类的父类是 Abstract-Action，但由于它的祖先 java.awt.event.ActionListener 是为定时器 javax.swing.Timer 所接受的接口，因此，java.awt.event.ActionListener 是抽象观察者，而 RotatingCursor 是具体观察者。

代码清单 3：TimerAction 类的源代码

```
package com.javapatterns.observertimer.cursor;
import javax.swing.AbstractAction;
import java.awt.event.ActionEvent;
import java.applet.Applet;
import java.awt.Cursor;
public class TimerAction extends AbstractAction
{
    private int ind = 0;
    private Applet applet;
   /**
    *  构造子
    */
    public TimerAction(Applet applet)
    {
        this.applet = applet;
    }
   /**
    *  事件方法
    */
    public void actionPerformed(ActionEvent e)
    {
        if (ind == 0)
        {
            drawCursor(Cursor.NE_RESIZE_CURSOR);
        }
        else if (ind == 1)
        {
            drawCursor(Cursor.N_RESIZE_CURSOR);
        }
        else if (ind == 2)
        {
            drawCursor(Cursor.NW_RESIZE_CURSOR);
        }
        else if (ind == 3)
        {
            drawCursor(Cursor.W_RESIZE_CURSOR);
```



```
            }
            else if (ind == 4)
            {
                drawCursor(Cursor.SW_RESIZE_CURSOR);
            }
            else if (ind == 5)
            {
                drawCursor(Cursor.S_RESIZE_CURSOR);
            }
            else if (ind == 6)
            {
                drawCursor(Cursor.SE_RESIZE_CURSOR);
            }
            else if (ind == 7)
            {
                drawCursor(Cursor.E_RESIZE_CURSOR);
                ind = -1;
            }

            ind++;
        }
        public void drawCursor(int cursorType)
        {
            applet.setCursor(Cursor.getPredefinedCursor(cursorType));
        }
    }
```

应当指出的是，虽然 Swing 定时器的构造子要求一个初始的观察者，但是 Swing 定时器同时提供一个如下的方法，使得客户端可以动态地增加观察者对象，如代码清单 4 所示。

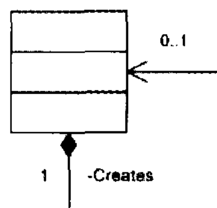
代码清单 4：addActionListener（ActionListener）方法

```
    /**
     * Adds an actionListener to the Timer
     */
    public void addActionListener(ActionListener listener)
    {
        ......
    }
```

因此，Swing 定时器的设计使每一个定时器对象可以有任意多的观察者对象。如果提供一个初始的观察者比较方便，那么就可以在调用构造子时提供，这样还可以省去以后再提供的麻烦；如果在调用构造子时提供初始观察者不方便，那么在调用构造子时可以传入 null 作为替代，然后通过调用 addActionListener（ActionListener）方法设置一个或多个观察者对象。

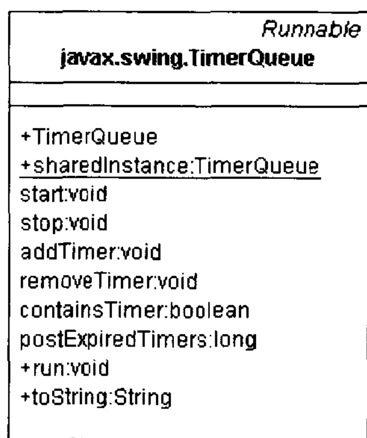
41.5 单例模式的应用

熟悉单例模式的读者一定会想，既然 Swing 定时器类使用单一的线程处理所有的定时器对象，那么这里面是不是使用了单例模式来实现这个单一性呢？单例模式的简略类图如右图所示。



TimerQueue 是一个单例类

对于上面的问题回答是肯定的。Swing 定时器类使用一个 TimerQueue 对象封装了等待线程，而且这个类是一个单例类。TimerQueue 类是一个单例类，如下图所示。



TimerQueue 类提供了一个静态方法 sharedInstance()，并通过这个方法提供惟一的一个实例。这个方法的源代码如代码清单 5 所示。

代码清单 5：静态方法 sharedInstance()向外界提供 TimerQueue 的惟一实例

```
public static TimerQueue sharedInstance()
{
    synchronized (classLock)
    {
        TimerQueue sharedInst = (TimerQueue)
            SwingUtilities.appContextGet(sharedInstanceKey);
        if (sharedInst == null)
        {
            sharedInst = new TimerQueue();
            SwingUtilities.appContextPut(sharedInstanceKey, sharedInst);
        }
        return sharedInst;
    }
}
```

是 Swing 的一个失误吗?

细心的读者可能会发现，TimerQueue 有一个公开的构造子，这个构造子将线程初始化，如代码清单 6 所示。

代码清单 6：构造子的源代码

```
/**
 * Constructor for TimerQueue.
 */
public TimerQueue()
{
    super();
    // Now start the TimerQueue thread.
    start();
}
```

根据单例模式的精神，一个单例类不应当有公开的构造子，所有的实例均应当由静态方法提供。提供公开的构造子会使外界对象通过调用此构造子创建多余的实例，从而破坏单例模式的用意。

事实上，如果对 Sun Microsystem 所提供的 JDK 1.3 的源代码做全文搜索的话，会发现只有以下三个类引用了 TimerQueue 类，而且全都是通过调用静态方法 sharedInstance()得到它的实例：

- JApplet
- SystemEventQueueUtilities
- Timer

比如在 Timer 类中，timerQueue()方法返回一个 TimerQueue 的实例，而这个方法是通过调用 TimerQueue 的静态方法 sharedInstance()达到的，其源代码如代码清单 7 所示。

代码清单 7：timerQueue()方法的源代码

```
/**
 * Returns the timer queue.
 */
TimerQueue timerQueue()
{
    return TimerQueue.sharedInstance();
}
```

因此，Swing 库包通过程序员自律达到了使用单例模式的目的，同时也佐证将构造子设成公开是不必要的。

回到本章所讨论的 Swing 定时器的话题，Swing 的 Timer 类实际上并不做任何的定时，它只是存储 TimerQueue 的实例和时间间隔的数值。Timer 类使用一个单例类 TimerQueue 保证了在整个 JVM 里面只有一个 TimerQueue 的实例，因此所有的 Timer 实例均共享同一个线程。

当调用 Timer 对象的 start()方法时，定时就开始了。如代码清单 8 所示，定时一旦开始，Timer 对象就把自己加入到 TimerQueue 里面。

代码清单 8：start()方法的源代码

```
public void start()
{
    timerQueue().addTimer(this,
        System.currentTimeMillis() + getInitialDelay());
}
```

在这以后，TimerQueue 对象会将所有的定时器对象按照触发时间排序，然后将它自身转入休眠状态，等待最近的一个触发时间的到来。当一个触发时间到来时，TimerQueue 会使所触发的定时器对象通知所有的观察者对象，如代码清单 9 所示。

代码清单 9：TimerQueue 对象调用 Timer 对象的 post()方法的源代码

```
if (timeToWait <= 0)
{
    try
    {
        timer.post();   // have timer post an event
    }
    catch (SecurityException e)
    {
    }
    // Remove the timer from the queue
    removeTimer(timer);
    ......
}
```

这时，Timer 对象调用 Swing 的 invokeLater()方法，并传入一个 Runnable 类型的对象，其 run()方法会发出一个 actionPerformed()信息给 Timer 对象的所有的 ActionListener 对象，在这里便是所有的 TimerAction 对象。

问答题

1. 请利用 Swing 定时器写一个浏览器小程序，实现数字钟的功能。
2. 请问第 1 题的系统里使用了什么模式。

问答题答案

1. 数字钟的运行情况如下图所示。





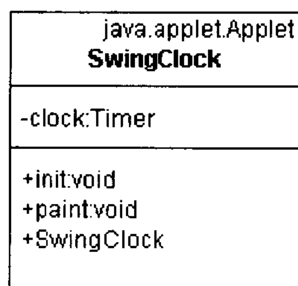
这个小程序的源代码如代码清单 10 所示。

代码清单 10：这个小程序的源代码

```
import javax.swing.Timer;
import java.text.DateFormat;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.Calendar;
import java.util.Date;
public class SwingClock extends java.applet.Applet
{
    private Timer clock = new Timer( 1000,
        new ActionListener()
        {
            public void actionPerformed( ActionEvent e )
            {
                synchronized( SwingClock.this )
                {
                    repaint();
                }
            }
        }
    );
    public void init()
    {
        setBackground(Color.green);
        setFont(new Font("TimesRoman",Font.BOLD,36));
    }
    public void paint(Graphics g)
    {
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        DateFormat dateFormatter = DateFormat.getTimeInstance();
```

```
        g.drawString(dateFormatter.format(date), 5, 100);
    }
    public SwingClock() throws InterruptedException
    {
        clock.start();
    }
}
```

小程序的类图如下图所示。



2. 在第 1 题的数字钟系统中使用了观察者模式。这个模式所涉及的角色有：

- （具体）被观察者角色：由定时器 javax.swing.Timer 类扮演。
- 具体观察者角色：由一个实现了 ActionListener 接口的无名内部类扮演。

参考文献

[WALRATH00] Hans Muller and Kathy Walrath. Using Timers in Swing Applications. http://java.sun.com/products/jfc/tsc/articles/timer/, 2000

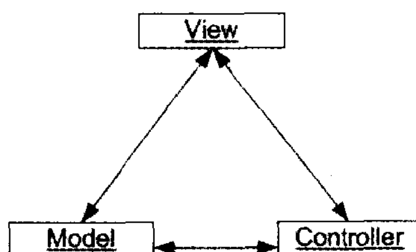
[HOLUB99] Allen Holub. Programming Java threads in the real world. JavaWorld (http://www.javaworld.com/javaworld/jw-02-1999), February 1999

第 42 章　专题：MVC 模式与用户输入数据检查

建议读者将此章与“观察者（Observer）模式”、“策略（Strategy）模式”，以及“合成（Composite）模式”一起阅读。

42.1　什么是 MVC 模式

所谓的 MVC 模式，即模型–视图–控制器（Model-View-Controller）模式。MVC 模式的结构图如下图所示。



MVC 模式的特殊性

MVC 模式常常作为一个设计模式出现在各种讨论中，但是 MVC 模式实际上是架构模式，而不是设计模式。

在 amazon.com 的专访“VLISSIDES98”中，John Vlissides 在他们的著作“GOF95”里提到：

“避免了可以分割成较小模式的大的模式。比如，我们没有把 Smalltalk 的模型–视图–控制器（Model-View-Controller，MVC）当做一个模式提出，因为它很大程度上是由我们已经有的较小的模式组成的。我承认，MVC 模式比单纯的合成模式加策略模式加观察者模式要多一些东西，但是我们不想在有更基本的模式需要讨论的时候，把时间花在讨论 MVC 模式上。”

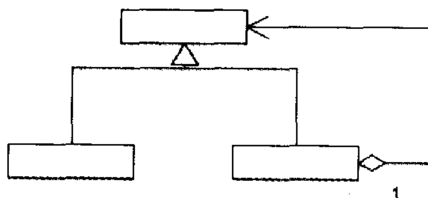
换言之，MVC 是一个比本书前面所讨论的设计模式更大的尺度上的模式，而不是本书所关注的尺度上的设计模式。

可以分解为几种设计模式

如果硬要把 MVC 模式与前面所讨论的设计模式对上号的话，那么它在大部分情况下是下面几个模式之一。

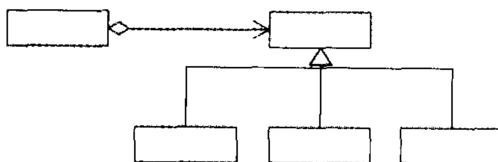
（1）合成模式

合成模式的简略类图如下图所示。



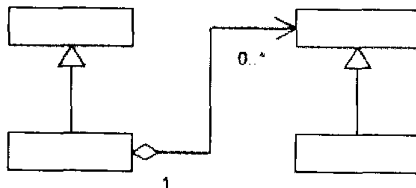
（2）策略模式

策略模式的简略类图如下图所示。



（3）观察者模式

观察者模式的简略类图如下图所示。



也有可能是三者的混合。它也会涉及到与以上三个模式有关的其他模式，比如：

- 合成模式：装饰模式、享元模式、迭代子模式、访问者模式
- 策略模式：享元模式
- 观察者模式：调停者模式、单例模式

请参考这三个模式的章节中“相关模式”一节。这就是说，如果把 MVC 模式当做设计模式来讨论的话，那么它会覆盖上面三个模式，以及它们的相关模式等，大约 10 个设计模式。

模式的种类

由于[GOF95]是论述软件模式的著作的第一本，也是 OO 设计理论著作中最流行的一本，因此有些人常常使用设计模式（Design Pattern）一词来指所有直接处理软件的架构、



设计、程序实现的任何种类的模式。另外一些人则强调要划分三种不同层次的模式：架构模式（Architectural Pattern）、设计模式（Design Pattern）、成例（Idiom）。成例有时称为代码模式（Coding Pattern）。

架构模式（Architectural Pattern）

一个架构模式描述软件系统里的基本的结构组织或纲要。架构模式提供一些事先定义好的子系统，指定它们的责任，并给出把它们组织在一起的法则和指南。有些作者把这种架构模式叫做系统模式“STELTING02”。

一个架构模式常常可以分解成很多个设计模式的联合使用。显然，MVC 模式就属于这一种模式。

设计模式（Design Pattern）

一个设计模式提供一种提炼子系统或软件系统中的组件或者它们之间的关系的纲要设计。设计模式描述普遍存在的在相互通信的组件中重复出现的结构，这种结构解决在一定的背景中的具有一般性的设计问题。

本书所讨论的二十几种模式，除 MVC 模式之外，都是设计模式。

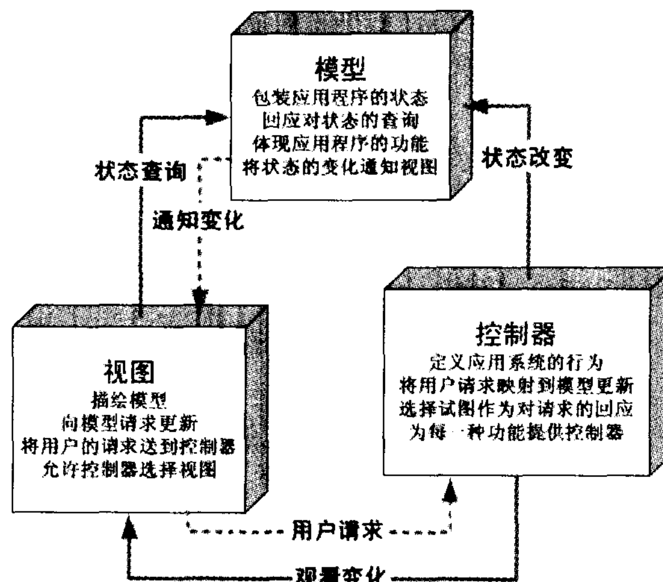
代码模式或成例（Coding Pattern 或 Idiom）

代码模式（或成例）是较低层次的模式，并与编程语言密切相关。代码模式描述怎样利用一个特定的编程语言的特点来实现一个组件的某些特定的方面或关系。

这三者之间的区别在于三种不同的模式存在于它们各自的抽象层次和具体层次上。架构模式是一个系统的高层次策略，涉及到大尺度的组件以及整体性质和力学。架构模式的好坏可以影响到总体布局和框架性结构。设计模式是中等尺度的结构策略。这些中等尺度的结构实现了一些大尺度组件的行为和它们之间的关系。模式的好坏不会影响到系统的总体布局和总体框架。设计模式定义出子系统或组件的微观结构。代码模式（或成例）是特定的范例和与特定语言有关的编程技巧。代码模式的好坏会影响到一个中等尺度组件的内部、外部的结构或行为的底层细节，但不会影响到一个部件或子系统的中等尺度的结构，更不会影响到系统的总体布局和大尺度框架。

42.2　MVC 是架构模式

在此将 MVC 模式划分为架构模式更为恰当。MVC 作为模式，提供了一个原则，可以按照模型、表达方式和行为等角色把一个应用系统的各个部分之间的耦合解脱、分割开来。MVC 模式的示意图如下图所示。



模型端

在 MVC 模型里，模型便是执行某些任务的代码，而这部分代码并没有任何逻辑决定它对用户端的表示方法。模型端只有纯粹的功能性接口，也就是一系列的公开方法。通过这些公开方法，便可以取得模型端的所有功能。

在这些公开方法中，有些是取值方法，让系统其他部分可以得到模型端的内部状态参数，其他的改值方法则允许外部修改模型端的内部状态。

但是，一般来说，模型端必须有方法登记视图，以便在模型端的内部状态发生变化时，可以通知视图端。在 Java 语言里，一个模型端可以继承 java.util.Observable 类。此父类可以提供登记和通知视图所需的接口。

多个视图端

在MVC模式里面，一个模型端可以有几个视图端，而实际上复数的视图端是使用MVC的原始动机。使用 MVC 模式可以允许多于一个的视图端存在，并且可以在需要的时候动态地登记上所需的视图。

举一个大家熟悉的例子，是 Excel 表格。一个饼图、一个棒图和一个表格均是同组数据的不同的视图端，当用户通过任何一个视图修改数据时，所有的视图都会按照新数据更新自己。

在 Java 语言的 java.awt 库和 java.swing 库里，所有的视窗构件均可用来建造视图端，但是一个视图如果能够自动得到更新，便需要实现 java.util.Observer 接口，这样便使得 MVC 模式符合观察者模式的定义。

在视图端里，视图可以嵌套，这意味着在视图端里会有合成模式。比如，在一个 Frame 构件里面，会有文字框构件和按键构件。

多个控制器端

MVC 模式的视图端是与 MVC 模式的控制器端结合使用的。当用户端与相应的视图发生交互时，用户可以通过视窗更新模型的状态，而这种更新是通过控制器端进行的。控制器端通过调用模型端的改值方法更改其状态值。与此同时，控制器端会通知所有的登记了的视图刷新显示给用户的表示。这意味着在视图端对象和控制器端对象之间会有观察者模式的应用。

一个控制器端对象在回应视图端请求时，会采用策略模式的方式决定如何回应。

读者可以阅读本书在“装饰模式”一章中给出的一个 MVC 模式的应用实例。

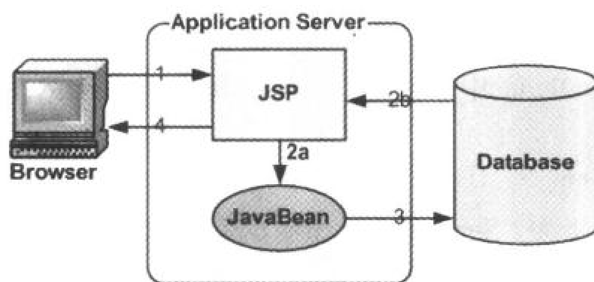
42.3　MVC 模式在 J2EE 技术中的应用

MVC 模式并不能自动保证一个结构设计是正确的，如何在一个系统的设计中正确地使用 MVC 架构模式与系统所使用的技术有密切的关系。比如在一个使用 JSP、Java Servlet 的 J2EE 系统中，如何正确地使用 MVC 模式就是一个值得研究的问题。

一般而言，一个 J2EE 系统应当适当地划分接收请求，根据请求采取行动，并将结果显示给用户等责任。流行的划分方式有两种，分别叫做 JSP 模型一和 JSP 模型二 [GOMEZ00]。

JSP 模型一的架构

模型一又称做以 JSP 为中心（JSP Centric）的设计模型，它的架构图如下图所示。



可以看出，其中的 JSP 负责与客户端通信，处理所有的请求和答复。数据库的存取直接由 JSP 完成，有时由一些 JavaBean 辅助完成。JavaBean 的用途是在不同的 JSP 之间通信。

在这个模型里，系统的活动时序如下所示：

（1）JSP 接到一个客户端请求并处理此请求。

（2）JSP 使用 JavaBean 读取在 Sesssion 对象中或者 Application 对象中共享的状态信息，或者是通过 JavaBean 存取数据库中的信息。

（3）JSP 也可以直接使用数据库中的信息，调用任何其他的 API。

（4）JavaBean 可以存取数据库中的信息。

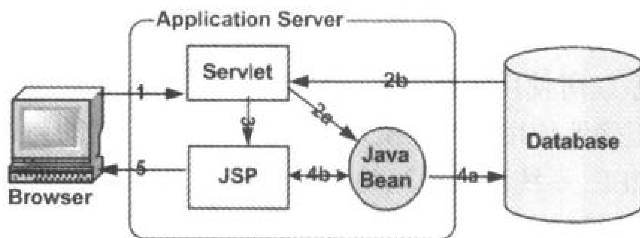
（5）JSP 可以将输出结果格式化为用户可以阅读的形式，并返还给客户端。

必须指出的是，显示数据的逻辑和数据在这个模型中有了一定程度的区分，但是商务逻辑是和显示数据的逻辑混合在 JSP 里面，使得两者无法独立演化。这就是模型一的缺点，这一缺点在模型二中得到了克服。

JSP 模型二的架构

模型二又叫做以 Servlet 为中心（Servlet Centric）的设计模型。

通常认为模型二是一个比模型一好很多的架构，因为它将显示数据的逻辑与商务逻辑分割开来，从而使得系统的层次更加清楚。由于商务逻辑和显示数据的逻辑是分开的，因此两者可以独立演化。JSP 模型二的架构图如下所示。



在这个模型里，系统的活动时序如下所示：

（1）Servlet 相当于控制器（Controller）角色，它负责接收客户端请求并处理此请求。

（2）根据请求的类型，Servlet 可以选择创建一个 JavaBean 对象，并从请求的处理过程中取得的结果作为初始化参数传给 JavaBean 对象。

（3）Servlet 也可以直接存取数据库中的数据。

（4）Servlet 将请求传递给合适的 JSP，而 JSP 则显示给用户。

（5）JSP 仅仅从 JavaBean 中读取数据，JavaBean 直接与数据库打交道。JSP 不与数据库打交道。

（6）JSP 返还给客户端。

可以看出，在模型二里面，Servlet 不参加显示数据的工作，从而没有显示数据的逻辑。Servlet 仅仅负责产生中间数据，将这些数据以 JavaBean 对象的形式存储在 Session 对象里面。

系统的可缩放性

必须指出的是，存储在 Session 对象中的 JavaBean 对象的大小直接影响到系统的可缩放性（scalability）。

这些 JavaBean 对象越大，系统的可缩放性就越差。特别是在使用 Application Server 的 clustering 功能时，这些对象会被复制到每一个参加 cluster 的服务器上，因此也会消耗



一定的资源。所以必须设法将这些 JavaBean 对象控制在尽可能小的程度上。

模型二对项目开发管理的好处

模型二虽然有消耗资源的缺点，但是它对项目开发管理带来的好处却远远大过它的缺点。

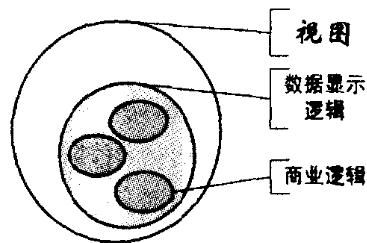
首先，尽管 JSP 会在第一次被调用时编译成为 Servlet，但是系统无法将这个动态生成出来的 Servlet 作为静态类型使用，大多数 Java 的 IDE 并不支持 JSP 的语法检查和差错功能。除非 JSP 被放到服务器去运行，不然往往无法看到错误。

其次，对于项目管理人员来说，模型二带来的好处也很明显，由于 JSP 不含有商务逻辑，因此不需要高水平的程序员来编程。JSP 可以交给专门做网页设计的人员去完成。

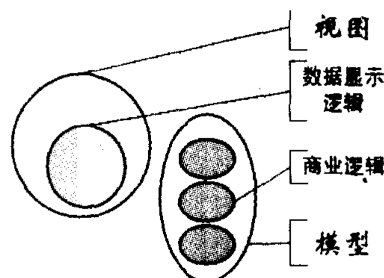
最后，由于 Servlet 和 JavaBean 含有所有的商务逻辑，因此项目管理人员可以根据团队成员的专业水平高低分配开发工作。

42.4　从代码重构的角度查看

在本节仍然以一个基于 Java 技术的动态网站系统为例进行说明。



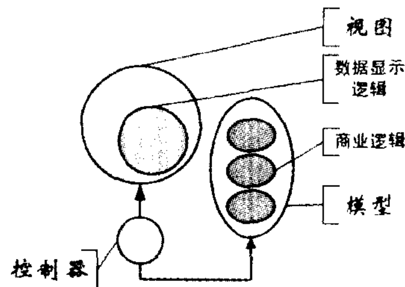
从代码重构的角度看，将 MVC 模式应用到一个系统设计中的过程就是对系统中不同责任的划分过程。假设一个系统使用 JSP 处理从数据库中取出数据，再使用数据显示逻辑将数据转化成对用户友好的格式。而在用户选中显示的数据并请求某种操作后，JSP 将控制转给下一张 JSP。使用 JSP 处理所有责任的概念图如右图所示。



作为架构设计的改良方案，应当将商业逻辑划分出来，形成一个或多个独立的模型角色。这样就需要创建一些 JavaBean，负责组成 SQL 语句并从数据库中取出数据，从而将数据显示逻辑与商业逻辑分割开来，划分到两个独立的角色中去，“视图角色”与“模型角色”将商业逻辑分化出来的概念图如右图所示。

这是一个典型的 JSP 模型二。而在很多情况下，一个设计可以到此为止了。但是也有一些情况，从一个 JSP 到另一个 JSP 的流程控制比较复杂，把这个逻辑混在商务逻辑或者数据显示逻辑里，会导致流程控制逻辑的变化影响到商务逻辑或者数据显示逻辑，或者相反。

因此，一个更好的架构设计应当将流程控制交给一个独立的角色管理，这个角色就是控制器角色。请参考其概念图，如下图所示。



在通常情况下，控制器角色由 Servlet 实现，其设计也可以借鉴命令模式的结构。

因此，在重构完成之后，JSP 会负责数据显示逻辑，JavaBean 负责商业逻辑，而 Servlet 负责流程的控制。看看上面的重构过程，像不像从一个单细胞生物到多细胞生物的演变过程？

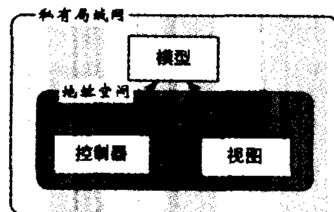
42.5 用户输入检查与 MVC 模式

下面讨论基于国际网络技术的信息系统的用户输入检查问题。

Client-Server 系统设计

在 Client-Server 系统设计中，一般倾向于将用户输入检查放到视图端。模型端和控制器端接收任何从视图端传来的用户输入数据。由于 Client-Server 系统的模型段通常就是数据库，而剩下的两层，即它的视图端和控制器端虽然可以在逻辑上分开，但往往是在同一个地址空间中运行的，物理上并不能分开。由于视图端和控制器端在物理上并不能分开，因此不存在数据传送过程中被恶意用户截获和篡改的可能性。

同时，由于模型端（数据库）与视图端和控制器端的联络是在防火墙内部的，因此模型端与控制器、模型与视图端的通信被截获和篡改的可能性也很小。Client-Server 系统的设计示意图如下图所示。

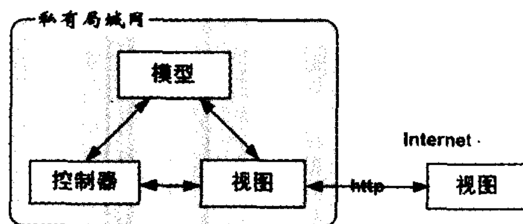


国际联网的网站系统

随着国际网络技术的普及，很多原先从事 Client-Server 系统设计的设计师也进入基于国际网络技术的信息系统的设计，他们往往不自觉地将很多 Client-Server 系统设计中的标准做法带入到基于国际网络技术的信息系统的设计中来。将用户输入检查的责任完全交给

视图端的做法也被很多人带入到后者的设计中，而这样会产生很多不安全的系统设计。

在一个基于国际网络技术的系统中，视图端的一部分往往在用户自己的浏览器（browser）内运行，另外的视图、控制器、模型端则是在服务器内部运行。在用户浏览器中的视图端与在服务器内的视图端通过 HTTP 协议经过国际网络相互联络。Client-Server 系统的设计示意图如下图所示。



这时候就需要特别注意数据可能被截获和篡改的问题。很多设计师错误地解释 MVC 模式，认为用户端输入检查是视图端的责任，从而将用户输入检查完全局限在客户端 JavaScript 检查上，而服务器端则接收任何传进来的用户输入数据，这是很危险的。

应用层的保护

使用 SSL 是保证通信安全的一个手段，使用 SSL 可以向用户证明服务器的身份，使用 SSL 可以保证通信是加密的。但是，使用 SSL 不能保证恶意的用户没有篡改用户输入的数据，SSL 也不能提供应用层的任何保护。

为了做到应用层的保护，本书建议读者使用下面的保护措施：

（1）使用带有密钥的 hashing 功能，比如 md5sum 进行数据完整性的核查。

（2）使用像 DES 这样的行业标准加密算法对数据进行加密（DES 即 Data Encryption Standard，是美国政府承认的加密算法标准，美国政府要求某些金融业电子化服务必须采取 DES 标准）。

（3）列出所有可能的用户端的无效输入，并在用户输入检查过程中加以排除。

必须记住，SSL 并不能提供以上的保护，SSL 保护数据在传送过程中的安全，但是并不能保护在浏览器中的数据元素。换言之，SSL 并不能保证用户输入数据没有被恶意用户利用，来窥视信息系统的秘密。

用户端输入检查

一个基于国际网络技术的信息系统的设计必须假设有一些用户是恶意的。这些用户可能利用系统的任何安全漏洞窥视系统的秘密，并利用系统的设计缺陷获取非法的商业价值。

为了保护信息系统的安全，设计师必须要做到以下两点：

（1）系统必须在服务器一端也进行基于安全考虑的用户输入数据的检查。虽然浏览器内部的 JavaScript 也可以用来对输入数据进行合法性检查，但是恶意用户可以使用诸如 Achilles（请参见本书“代理模式”一章）等软件将传到客户端的网页做相应的修改，再

传到浏览器上。这样恶意用户可以将妨碍其恶意操作的 JavaScript 轻易地去掉。

（2）不可将系统内部出错的信息传给客户端，不可以将含有详细程序错误信息的出错信息传给浏览器。没有一个系统是完美无缺的，任何系统都会出错，特别是在恶意用户蓄意制造出错的情况下。在出错的情况下，仅给客户端显示一个抱歉信息，诸如“很抱歉，系统出现运行错误，请和某某联系”一类的出错信息，而绝对不应当给出具体的 JVM 错误信息。

换言之，出错信息不可以原封不动地传递给位于浏览器内部的视图端。

42.6 SQL 注射

为了说明上面所提出的两项预防措施的必要性，这里特别给出一个臭名昭著的恶意用户攻击网络信息系统的称为“SQL 注射”的手法。《孙子兵法》说，“知彼知己，百战不殆”，了解对手的手段便是设计一个安全系统的出发点。

什么是 SQL 注射

所谓 SQL Injection，也就是 SQL 注射，又叫做 SQL Piggybacking。这是一种利用一些系统没有足够的用户输入数据检查的缺点，而在用户输入数据中“注射”进非法 SQL 语句段，利用信息系统的 SQL 引擎进行自己的信息查询工作的恶意做法。这一做法的特点如下：

（1）通过小心地构造用户输入，将数据库命令注射进用户端输入的数据中去，进而操控后台数据库。

（2）这一做法非常依赖于特定的技术环境和技术。

（3）这是一个尝试–错误–再尝试的过程，因此任何的出错信息都对恶意用户非常有用。

恶意用户可以通过 SQL 注射做什么

通过这一做法，具有高级信息系统知识的恶意用户可以做到以下几点：

（1）可以用来提出本来没有权限取得的信息。

（2）改变敏感数据。

（3）删除数据库中的数据，甚至表。

SQL 注射的过程

那么恶意用户是怎么利用 SQL 注射对网络信息系统进行攻击的呢？

（1）恶意用户会寻找可能成为某个数据库 SQL 语句的一部分的用户输入文字框，比如用户名、账号号码、产品序列号码、住址等。



（2）恶意用户将单引号（'）或者双引号（"）加入到文字框中去。如果网页有客户端的 JavaScript 检查，那么就使用像 Achilles 这样的工具（参见本书“代理模式”一章）将 JavaScript 函数取消。这样，恶意用户就可以将这些带有特殊字符的数据传给信息系统了。

恶意用户可能使用的特殊字符包括：

- 冒号（:）：是一些数据库语言的语句结束符号。
- 星号（*）：在一些数据库语言中表示默认匹配。
- 百分号（%）：在一些数据库语言中表示默认匹配。
- 下划线（_）：在一些数据库语言中表示字符的默认匹配。

恶意用户可以使用通常的 SQL 语句，例如：

```
select * from <table> where <column> = <value>
```

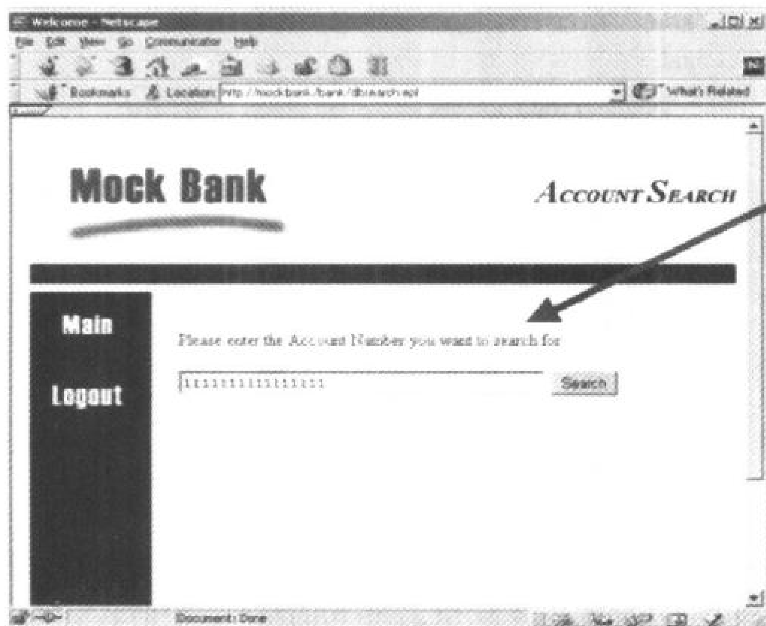
或者

```
update <table> set <column> = <value>
```

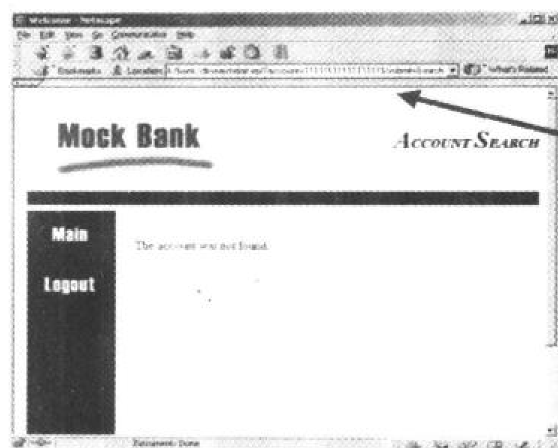
等，将之注射进信息系统所接收的用户输入数据中。

第一步：锁定目标

比如，恶意用户选定一个供用户输入账号号码的文字框。首先，恶意用户会使用一个不太可能存在的账号号码进行试探，恶意用户选择了一个供用户输入账号号码的文字框，如下图所示。



这时系统的反应是给出一个“账号并不存在”的信息。没关系，这正是恶意用户所期待的。系统对假账号号码的反应如下图所示。

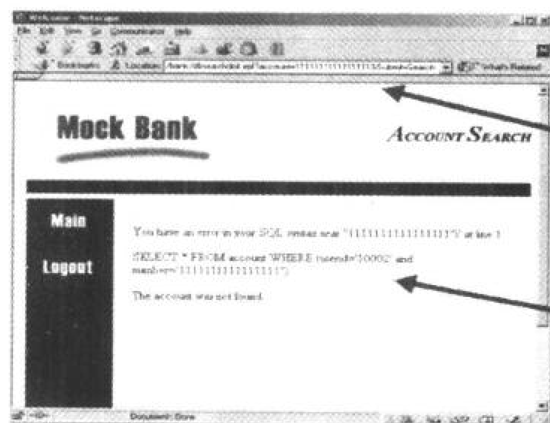


在这个例子里，恶意用户注意到账号号码出现在浏览器的 URL 框里面，因此恶意用户可以不理睬前面网页所提供的文字框，而直接操作 URL 框，从而根本就无需使用 Achilles 一类软件，绕开 JavaScript 函数的检查。

第二步：测试特殊字符

恶意用户接下来就在账号号码中加入一个单引号（'），试探系统对特殊字符的反应，如下图所示。这时，本系统的反应是给出如下的系统内部错误信息：

You have an error in your SQL syntax near "111111111111")' at line 1
SELECT * FROM account WHERE (userid = '10002' and
number='111111111111')
The account was not found.



从上面的图中可以看出，系统竟然将完整的程序出错信息提供给客户端了！恶意用户大喜过望，因为恶意用户从此就知道了 SQL 语句的结构，表的名字（account）和列的名字（userid 和 number），下面会开始真正的攻击了。

第三步：SQL 注射与信息查询

这时候，系统的数据库就变成恶意用户随时可以免费获取的信息库了，如下图所示。



这时候，恶意用户在账号号码框中输入：

1111111111111111111' OR userid='10001

当数据库接到数据库命令时，这个命令成了：

SELECT * FROM account WHERE number='1111111111111111111' OR userid='10001'

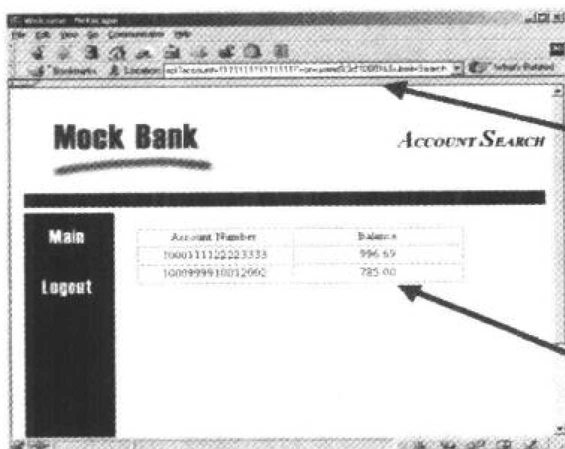
注意，系统本来期待的输入是：

SELECT * FROM account WHERE number='1111111111111111111'

而下面的 SQL 语句片断被“注射”进系统所期待的语句中：

' OR userid='10001

运行结果如下图所示。



教训总结：首先，系统的出错信息不可以直接提供给视图端，视图端应当只给出一个非常一般化的信息，而不应当给出任何与错误有关的具体信息；其次，用户输入数据检查功能不能仅局限于浏览器中的视图端，服务器端也必须有严密的用户输入数据检查功能。

MVC 模式作为一种架构模式而不是设计模式在系统的架构设计中有很重要的应用。MVC 模式可以用几种设计模式来实现。MVC 模式在 J2EE 技术中的应用应当选择以 Servlet 为中心的设计。

MVC 模式的使用不能够成为放松系统安全性能的理由，服务器端必须有严密的用户输入数据检查功能。视图端不应当原封不动地显示系统的具体出错信息，而应当给出一个非常一般化的出错信息。

问答题

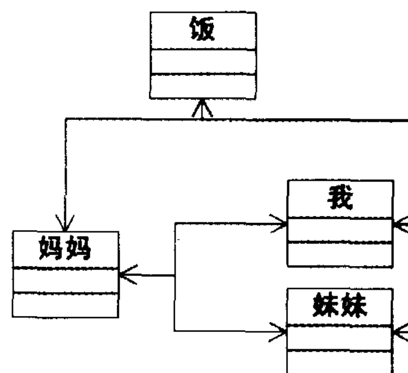
1. 我跟妈妈说：“妈妈，我和妹妹在院子里玩，饭做好了叫我们一声。”请问这是什么模式？

2. 有人说，“模式是一种规则，可大可小。比如说 MVC，可大到一个系统的架构，也可小到 Java 中的 JTable。用与不用在于具体的情况。”（选自 www.umlchina.com 的一段帖子）请问这种说法对吗？

3. 读过了本章关于 SQL 注射的介绍后，我觉得使用客户端 JavaScript 函数（即在浏览器中运行的 JavaScript 函数）进行输入数据检查就变得没有意义了。是这样吗？

4. 读过了本章关于 SQL 注射的介绍后，我觉得在名字和电子邮件地址中应当禁止使用单引号（'），这样可以吗？

问答题答案



1. 这是 MVC 模式。我和妹妹让妈妈告诉我们饭做好了，这样我们就可以来吃饭了。换用技术化的语言来说，当系统的模型端（饭）发生变化时，控制器端（妈妈）就告诉系统的视图端，也就是我和妹妹，使得它们可以调整内部状态（有开始吃饭的准备），并采取相应的行动（吃饭）。模拟系统的结构图如右图所示。

当然仔细分析起来，这里面还会有观察模式、合成模式甚至策略模式。

2. MVC 是架构模式，确实是可大可小。大可用到一个大型系统的架构，小可用到一个视窗上。从这一点上来讲，这个帖子是对的。

但是，设计模式与架构模式不同。设计模式有固定的应用尺度，不仅仅是可大可小的原则而已。

3. 不是。使用客户端 JavaScript 函数做输入数据检查仍然是必要的。如果所有的输入数据检查都必须在服务器端进行的话，会造成不必要的时间和资源上的消耗。

客户端 JavaScript 函数仍然是进行输入数据检查的很好工具，只是不应当把它当做惟一的数据检查。

4. 如果系统的用户限于国内用户的话，可以禁止在姓名框和电子邮件框中使用单引号。如果系统用户可能有国外用户的话，禁止在姓名框和电子邮件框中使用单引号就不可以。

一个简单的例子就是一些爱尔兰姓氏带有单引号，如 O'Neill 就是一个常见的姓氏，而 david.o'neill@abc.com 就是合法的电子邮件地址。

参考文献

[VLISSIDES98]John Vlissides. Amazon.com Interview. http://www.research.ibm.com/designpatterns/pubs/amazon-interview.htm.

[STELTING02]Stephen Stelting and Olav Maassen. Applied Java Patterns. Prentice Hall, 2000

[GOMEZ00] Paco Gomez and Peter Zadrozny. Professional Java 2 Enterprise Edition with BEA WebLogic Server. Wrox Press, 2000

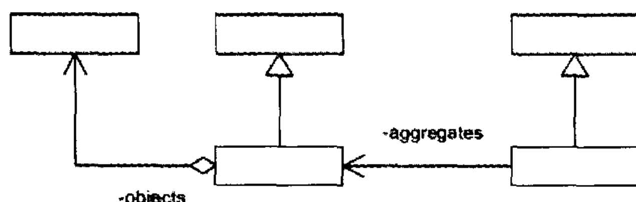
第 43 章　迭代子（Iterator）模式

迭代子（Iterator）模式又叫游标（Cursor）模式[GOF95]，是对象的行为模式。迭代子模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部表象。

43.1　引　言

什么是迭代子模式

迭代子模式的简略类图如下图所示。



迭代子模式是文献[GOF95]提出的设计模式中最常见的几个设计模式之一，也是被广泛地应用到 Java 语言 API 中的几个设计模式之一。在 Java 语言的聚集（Collection）框架中，广泛使用迭代子来遍历聚集的元素。

所有熟悉 Java 语言的读者都应当了解迭代子在 Java 聚集中的应用，因此本章就从聚集讲起。在下一章“专题：Java 对迭代子模式的支持”会进一步讨论迭代子模式在 Java 聚集中的应用。

聚集和 Java 聚集

多个对象聚在一起形成的总体称之为聚集（Aggregate），聚集对象是能够包容一组对象的容器对象。聚集依赖于聚集结构的抽象化，具有复杂性和多样性。数组就是最基本的聚集，也是其他的 Java 聚集对象的设计基础。

Java 聚集（Collection）对象是实现了共同的 java.util.Collection 接口的对象，是 Java 语言对聚集概念的直接支持。从 1.2 版开始，Java 语言提供了很多种聚集，包括 Vector、ArrayList、Queue、Stack、LinkedList、HashSet、TreeSet、Hashtable、Hashmap 以及 TreeMap 等，这些都是 Java 聚集的例子。

当本书提到聚集时，是泛指包括 Java 聚集在内的一般性的对象集合；而当本书提到 Java 聚集时，则专指实现了 java.util.Collection 接口的那些聚集对象。

为什么聚集需要迭代子

聚集对象必须提供适当的方法，允许客户端能够按照一个线性顺序遍历所有的元素对象，把元素对象提取出来或者删除掉等。一个使用聚集的系统必然会使用这些方法操控聚集对象，因而在使用聚集的系统演化过程中，会出现两类问题：

（1）迭代逻辑没有改变，但是需要将一种聚集换成另一种聚集。因为不同的聚集具有不同的遍历接口，所以需要修改客户端代码，以便将已有的迭代调用换成新聚集对象所要求的接口。

（2）聚集不会改变，但是迭代方式需要改变。比如原来只需要读取元素和删除元素，但现在需要增加新的元素；或者原来的迭代仅仅遍历所有的元素，而现在则需要对元素加以过滤等。这时就只好修改聚集对象，修改已有的遍历方法，或者增加新的方法。

显然，出现这种情况是因为所涉及的聚集设计不符合“开–闭”原则，也就是因为没有将不变的结构从系统中抽象出来，与可变成分分割，并将可变部分的各种实现封装起来。一个聪明的做法无疑是应当使用更加抽象的处理方法，使得在进行迭代时，客户端根本无需知道所使用的聚集是哪个类型；而当客户端需要使用全新的迭代逻辑时，只需引进一个新的迭代子对象即可，根本无需修改聚集对象本身。

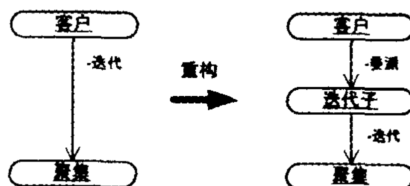
迭代子模式便是这样的一个抽象化的概念。这一模式之所以能够做到这一点，是因为它将迭代逻辑封装到一个独立的迭代子对象中，从而与聚集本身分割开。迭代子对象是对遍历的抽象化，不同的聚集对象可以提供相同的迭代子对象，从而使客户端无需知道聚集的底层结构。一个聚集可以提供多个不同的迭代子对象，从而使得遍历逻辑的变化不会影响到聚集对象本身。

“开–闭”原则

“开–闭”原则要求系统可以在不修改已有代码的情况下进行功能的扩展，做到这一点的途径就是对变化的封装。关于设计原则的讨论，请参见本书的“开–闭原则（OCP）”一章。

从对变化的封装角度讲，迭代子模式将访问聚集元素的逻辑封装起来，并且使它独立于聚集对象的封装。这就提供了聚集存储逻辑与迭代逻辑独立演变的空间，增加了系统的可复用性。这样系统更加符合“开–闭”原则的要求，可以使系统具有在无需修改的情况下进行扩展的能力。

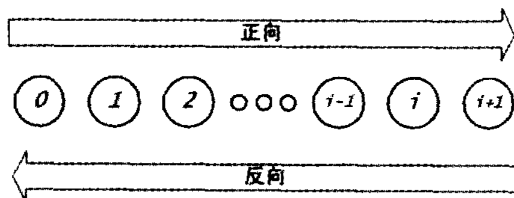
从代码重构的角度上讲，迭代子在客户端和聚集之间增加了一个中介层，从而使得客户端与聚集之间的通信从直接变成间接。这样做的好处就是缓冲了客户端的变化对聚集的影响，以及聚集的变化对客户端的影响。从代码重构的角度看如右图所示。





用词上的约定

为了避免用词上的混淆，本章首先对一些术语加以约定。每当本书提到一个序列时，将序列的元素指标增加的方向定为正方向（Forward），减少的方向定为反方向（Backward），有时也可称为正向和逆向。正向和反向的表示如下图所示。



所以，当本书说一个元素 A 位于元素 B 的后面时，意味着 A 的指标小于 B 的指标；反之，如果一个元素 A 位于元素 B 的前面时，意味着 A 的指标大于 B 的指标。一个元素的指标为 i 时，$i-1$ 位于它的后面（After），而 $i+1$ 位于它的前面（Before）。有的时候也可以说 $i-1$ 是 i 的上一个元素，$i+1$ 是 i 的下一个元素。前面和后面表示如下图所示。



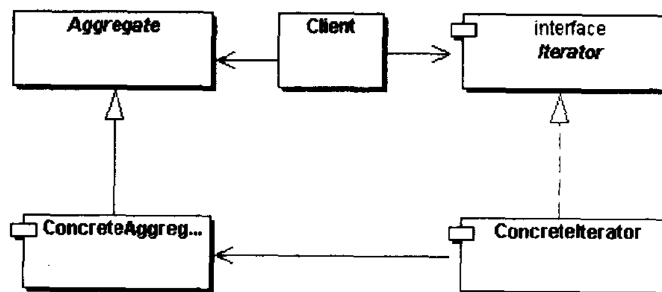
按照迭代的方向，迭代可以分成两种：一种为正迭代（Forward Iteration），一种为逆迭代（Backward Iteration）。正迭代按照一个列的正方向进行迭代；而逆迭代是按照一个列的反方向进行迭代。

43.2　迭代子模式的结构

迭代子模式如何实现影响它的结构的细节，因此，本章首先给出此模式的一般性结构，然后讨论模式的实现与相应的结构问题。

一般性结构

迭代子模式的一般性结构图如下图所示。



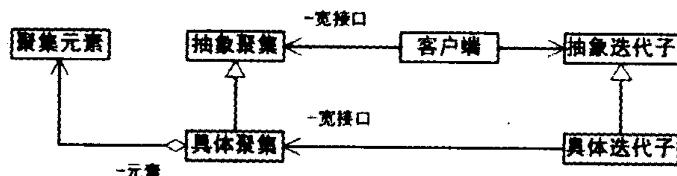
迭代子模式涉及到以下几个角色：

- 抽象迭代子（Iterator）角色：此抽象角色定义出遍历元素所需的接口。
- 具体迭代子（ConcreteIterator）角色：此角色实现了 Iterator 接口，并保持迭代过程中的游标位置。
- 聚集（Aggregate）角色：此抽象角色给出创建迭代子（Iterator）对象的接口。
- 具体聚集（ConcreteAggregate）角色：实现了创建迭代子（Iterator）对象的接口，返回一个合适的具体迭代子实例。
- 客户端（Client）角色：持有对聚集及其迭代子对象的引用，调用迭代子对象的迭代接口，也有可能通过迭代子操作聚集元素的增加和删除。

宽接口和窄接口

如果一个聚集的接口提供了可以用来修改聚集元素的方法，这个接口就是所谓的宽接口；如果一个聚集的接口没有提供修改聚集元素的方法，这样的接口就是所谓的窄接口。

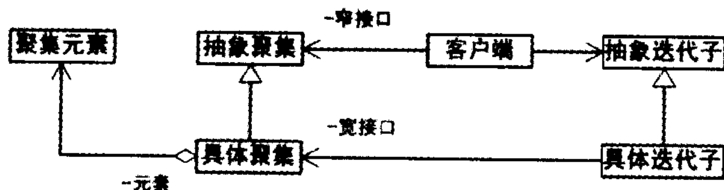
如果聚集对象为所有对象提供同一个接口，也就是宽接口的话，当然会满足迭代子模式对迭代子对象的要求。但是，这样会破坏对聚集对象的封装。这种提供宽接口的聚集叫做白箱聚集。聚集对象向外界提供同样的宽接口，如下图所示。



由于聚集自己实现迭代逻辑，并向外部提供适当的接口，使得迭代子可以从外部控制聚集元素的迭代过程。这样一来迭代子所控制的仅仅是一个游标而已，这种迭代子叫做游标迭代子（Cursor Iterator）。

由于迭代子是在聚集的结构之外的，因此这样的迭代子又叫做外禀迭代子（Extrinsic Iterator）。

在改良的设计中，聚集对象为迭代子对象提供一个宽接口，而为其他对象提供一个窄接口。换言之，聚集对象的内部结构应当对迭代子对象适当公开，以便迭代子对象能够对聚集对象有足够的了解，从而可以进行迭代操作。但是，聚集对象应当避免向其他的对象提供这些方法，因为其他对象应当经过迭代子对象进行这些工作，而不是直接操控聚集对象。在备忘录模式中，有类似的双重接口的做法，感兴趣的读者可以参阅本书的“备忘录（Memento）模式”一章。聚集对象向外界提供两种不同的接口，如下图所示。



在 Java 语言中，实现双重接口的办法就是将迭代子类设计成聚集类的内部成员类。这样迭代子对象将可以像聚集对象的内部成员一样访问聚集对象的内部结构。本节下面就给出一个示意性的实现，说明这种双重接口的结构是怎样产生的，以及使用了双重接口结构之后迭代子模式的实现方案。这种同时保证聚集对象的封装和迭代子功能的实现的方案叫做黑箱实现方案。

由于迭代子是聚集的内部类，迭代子可以自由访问聚集的元素，所以迭代子可以自行实现迭代功能并控制对聚集元素的迭代逻辑。由于迭代子是在聚集的结构之内定义的，因此这样的迭代子又叫做内禀迭代子（Intrinsic Iterator）。

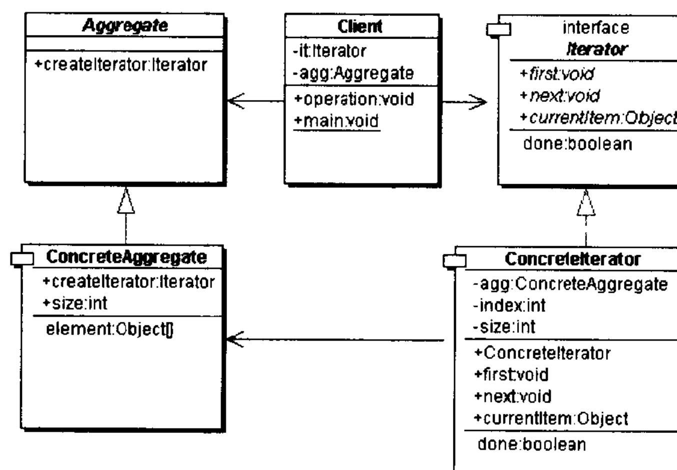
关于内禀、外禀迭代子与一些文献所定义的内部（Internal）、外部（External）迭代子的区别，请参见本章在后面“迭代子模式的实现”一节中给出的讨论。

白箱聚集与外禀迭代子

首先看一看白箱聚集与外禀迭代子的实现。一个白箱聚集向外界提供访问自己内部元素的接口（称做遍历方法或者 Traversing Method），从而使外禀迭代子可以通过聚集的遍历方法实现迭代功能。

因为迭代的逻辑是由聚集对象本身提供的，所以这样的外禀迭代子角色往往仅仅保持迭代的游标位置。

一个典型的由白箱聚集和外禀迭代子组成的系统如下图所示，在这个实现中具体迭代子角色是一个外部类，而具体聚集角色向外界提供遍历聚集元素的接口。



首先来看一看抽象聚集角色的源代码，如代码清单 1 所示，这个角色规定出所有的具体聚集必须实现的接口。迭代子模式要求聚集对象必须有一个工厂方法，也就是 createIterator()方法，以向外界提供迭代子对象的实例。

代码清单 1：抽象聚集角色 Aggregate 的源代码

```
package com.javapatterns.iterator.whitebox;
abstract public class Aggregate
{
```

```
    /**
     *  工厂方法：返还一个迭代子对象
     */
    public Iterator createIterator()
    {
        return null;
    }
}
```

抽象迭代子角色声明了具体迭代子需要实现的接口，也就是提供迭代功能的各个方法，有时也叫做迭代方法（Iteration Method），其源代码如代码清单 2 所示。

代码清单 2：抽象迭代子角色 Iterator 接口的源代码

```
package com.javapatterns.iterator.whitebox;
public interface Iterator
{
    /**
     *  迭代方法：移动到第一个元素
     */
    void first();
    /**
     *  迭代方法：移动到下一个元素
     */
    void next();
    /**
     *  迭代方法：是否是最后一个元素
     */
    boolean isDone();
    /**
     *  迭代方法：返还当前元素
     */
    Object currentItem();
}
```

具体聚集角色实现了抽象聚集角色所要求的接口，也就是 createIterator()方法。此外，还有遍历方法 getElement()向外界提供聚集元素，而遍历方法 size()向外界提供聚集的大小等，如代码清单 3 所示。

代码清单 3：具体聚集角色 ConcreteAggregate 的源代码

```
package com.javapatterns.iterator.whitebox;
public class ConcreteAggregate extends Aggregate
{
    private Object[] obj = {"Monk Tang",
        "Monkey", "Pigsy",
        "Sandy", "Horse"};
    /**
     *  工厂方法：返还一个迭代子对象
```



```
    */
    public Iterator createIterator()
    {
        return new ConcreteIterator(this);
    }
    /**
     *  取值方法：向外界提供聚集元素
     */
    public Object getElement(int index)
    {
        if (index < objs.length)
        {
            return objs[index];
        }
        else
        {
            return null;
        }
    }
    /**
     *  取值方法：向外界提供聚集的大小
     */
    public int size()
    {
        return objs.length;
    }
}
```

可以看出，这个具体聚集对象封装了一个数组，数组的元素在类被加载时就被初始化了。

为简化起见，在这个实现里聚集对象是一个不变对象。如果一个对象的内部状态在对象被创建之后就不再变化，这个对象就称为不变对象。如果一个聚集对象的内部状态可以改变的话，那么在迭代过程中，一旦聚集元素发生改变（比如一个元素被删除，或者一个新的元素被加进来），就会影响到迭代过程，使迭代无法给出正确的结果。关于这一点的讨论，请参见本章的“迭代子模式的实现”一节；而关于不变模式的讨论，请参见本书的“不变（Immutable）模式”一章。

迭代子模式并不要求聚集对象是不变对象，是否将聚集设计成为不变对象仅仅是一个具体实现的问题，如代码清单 4 所示，设计师完全可以根据系统需要决定。

代码清单 4：具体迭代子角色 ConcreteIterator 类的源代码

```
package com.javapatterns.iterator.whitebox;
public class ConcreteIterator implements Iterator
{
    private ConcreteAggregate agg;
    private int index = 0;
```

```
    private int size = 0;
   /**
    *  构造子
    */
    public ConcreteIterator(ConcreteAggregate agg)
    {
        this.agg = agg;
        size = agg.size();
        index = 0 ;
    }
   /**
    *  迭代方法：移动到第一个元素
    */
    public void first()
    {
        index = 0 ;
    }
   /**
    *  迭代方法：移动到下一个元素
    */
    public void next()
    {
        if (index < size)
        {
            index++;
        }
    }
   /**
    *  迭代方法：是否是最后一个元素
    */
    public boolean isDone()
    {
        return (index >= size);
    }
   /**
    *  迭代方法：返还当前元素
    */
    public Object currentItem()
    {
        return agg.getElement(index);
    }
}
```

可以看出，具体迭代子的构造子接收一个具体聚集对象作为参量，这就使得外禀迭代子可以根据这一引用控制聚集对象。

这个系统的一个示意性客户类的源代码，如代码清单 5 所示。

代码清单 5：客户角色 Client 类的源代码

```
package com.javapatterns.iterator.whitebox;
public class Client
{
    private Iterator it;
    private Aggregate agg = new ConcreteAggregate();
    public void operation()
    {
        it = agg.createIterator();
        while( !it.isDone() )
        {
            System.out.println(it.currentItem().toString());
            it.next();
        }
    }
    public static void main(String[] args)
    {
        Client client = new Client();
        client.operation();
    }
}
```

可以看出，上面这个示意性的客户端首先创建了一个聚集类的实例，然后调用聚集对象的工厂方法 createIterator()以得到一个迭代子对象。

在得到迭代子的实例后，客户端开始迭代过程，打印出所有的聚集元素，这表明迭代子有能力进行针对聚集的迭代。

外禀迭代子的意义

一个常常会问的问题是：既然白箱聚集已经向外界提供了遍历方法，客户端已经可以自行进行迭代了，为什么还要应用迭代子模式，并创建一个迭代子对象进行迭代呢？

客户端当然可以自行进行迭代，不一定非得需要一个迭代子对象。但是，迭代子对象和迭代模式会将迭代过程抽象化，将作为迭代消费者的客户端与迭代负责人的迭代子责任分隔开，使得两者可以独立演化。在聚集对象的种类发生变化，或者迭代的方法发生改变时，迭代子作为一个中介层可以吸收变化的因素，而避免修改客户端或者聚集本身。

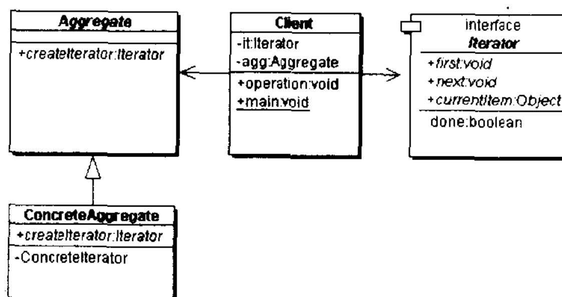
此外，如果系统需要同时针对几个不同的聚集对象进行迭代，而这些聚集对象所提供的遍历方法有所不同时，使用迭代子模式和一个外界的迭代子对象是有意义的。具有同一迭代接口的不同迭代子对象处理具有不同遍历接口的聚集对象，使得系统可以使用一个统一的迭代接口进行所有的迭代。

黑箱聚集与内禀迭代子

一个黑箱聚集不向外部提供遍历自己元素对象的接口，因此，这些元素对象只可以被

聚集内部成员访问。由于内禀迭代子恰好是聚集内部的成员子类，因此，内禀迭代子对象是可以访问聚集的元素的。

为了说明黑箱方案的细节，本节在这里给出一个示意性的黑箱实现，如下图所示。在这个实现里，具体聚集类 ConcreteAggregate 含有一个内部成员类 ConcreteIterator，也就是实现了抽象迭代子接口的具体迭代子类，同时聚集并不向外界提供访问自己内部元素的方法。



首先来看一看抽象聚集角色的源代码，如代码清单 6 所示。这个角色规定除所有的具体聚集必须实现的接口外，迭代子模式要求聚集对象必须有一个工厂方法，也就是 createIterator()方法，以提供迭代子对象的实例。

代码清单 6：抽象聚集角色 Aggregate 的源代码

```
package com.javapatterns.iterator.blackbox;
abstract public class Aggregate
{
    /**
     *  工厂方法：返还一个迭代子对象
     */
     public abstract Iterator createIterator();
}
```

抽象迭代子角色声明了具体迭代子需要实现的接口，也就是提供迭代功能的各个方法，如代码清单 7 所示。

代码清单 7：抽象迭代子角色 Iterator 接口的源代码

```
package com.javapatterns.iterator.blackbox;
public interface Iterator
{
    /**
     *  迭代方法：移动到第一个元素
     */
     void first();
    /**
     *  迭代方法：移动到下一个元素
     */
     void next();
    /**
```

```
     *  迭代方法：是否是最后一个元素
     */
    boolean isDone();
   /**
     *  迭代方法：返还当前元素
     */
    Object currentItem();
}
```

具体聚集角色实现了抽象聚集角色所要求的接口，也就是 createIterator()方法。此外，聚集类有一个内部成员类 ConcreteIterator，这个内部类实现了抽象迭代子角色所规定的接口；而工厂方法 createIterator()所返还的就是这个内部成员类的实例，如代码清单 8 所示。

代码清单 8：具体聚集角色 ConcreteAggregate 的源代码

```
package com.javapatterns.iterator.blackbox;
public class ConcreteAggregate extends Aggregate
{
    private Object[] obj = {"Monk Tang",
        "Monkey", "Pigsy",
        "Sandy", "Horse"};
   /**
     *  工厂方法：返还一个迭代子对象
     */
    public Iterator createIterator()
    {
        return new ConcreteIterator();
    }
   /**
     *  内部成员类：具体迭代子类
     */
    private class ConcreteIterator
        implements Iterator
    {
        private int currentIndex = 0;
       /**
         *  迭代方法：移动到第一个元素
         */
        public void first()
        {
            currentIndex = 0;
        }
       /**
         *  迭代方法：移动到下一个元素
         */
        public void next()
        {
```

```
                if ( currentIndex < obj.length )
                {
                    currentIndex++;
                }
            }
            /**
            *  迭代方法：是否是最后一个元素
            */
            public boolean isDone()
            {
                return (currentIndex == obj.length);
            }
            /**
            *  迭代方法：返还当前元素
            */
            public Object currentItem()
            {
                return obj[currentIndex];
            }
        }
    }
```

可以看出这个具体聚集对象封装了一个数组，数组的元素是类被加载时就被初始化的。在被加载后，聚集的元素就不再变化。这仅仅是为展示设计思想而对问题进行的简化，并不意味着迭代子模式只能处理聚集元素不发生变化的情况。这个具体聚集对象本身没有向外界提供遍历方法。

一个示意性的客户类的源代码如代码清单 9 所示。

代码清单 9：客户角色 Client 类的源代码

```
package com.javapatterns.iterator.blackbox;
public class Client
{
    private Iterator it;
    private Aggregate agg = new ConcreteAggregate();
    public void operation()
    {
        it = agg.createIterator();
        while( !it.isDone() )
        {
            System.out.println(it.currentItem().toString());
            it.next();
        }
    }
    public static void main(String[] args)
    {
        Client client = new Client();
```



```
        client.operation();
    }
}
```

可以看出，上面这个示意性的客户端首先创建了一个聚集类的实例，然后调用聚集对象的工厂方法 createIterator()，以得到一个迭代子对象的实例。在得到迭代子实例后，客户端开始迭代过程，打印出所有的聚集元素，这表明迭代子有能力进行针对聚集的迭代。

白箱聚集可以是不变对象吗

一个白箱聚集对象可能是不变对象，前提是它的内部状态，包括聚集元素是不会改变的。

一个白箱聚集对象自行向外界提供遍历方法，而外禀迭代子对象必须存储迭代的游标，因此，在迭代过程中白箱聚集本身并不存储游标，所以它的状态是可以保持不变的。

黑箱聚集可以是不变对象吗

一个黑箱聚集对象不可能是不变对象，因为它的内部状态是会改变的。

一个黑箱聚集对象自行向外界提供一个内禀迭代子对象，而迭代子对象必须存储迭代的游标，因此，在迭代过程中迭代子对象的状态是会改变的。这就意味着聚集对象的状态也是会改变的。

关于不变模式的讨论，请参阅本书的“不变（Immutable）模式”一章。

43.3 迭代子模式的实现

迭代子模式的实现可以很复杂。在本章前面的“迭代子模式的结构”一节里，已经涉及到迭代子模式在实现上的两种可能：外禀迭代子和内禀迭代子。

主动迭代子和被动迭代子

迭代子是主动的（Active）还是被动的（Passive），是相对于客户端而言的。如果客户端控制迭代的进程，那么这样的迭代子就是主动迭代子；相反就是被动迭代子。

使用主动迭代子的客户端会明显调用迭代子的 next()等迭代方法，在遍历过程中向前行进；而客户端在使用被动迭代子时，客户端并不明显地调用迭代方法，迭代子自行推进遍历过程。

何时使用内禀迭代子和外禀迭代子

在文献[GOF95]中，外部迭代子（External Iterator）和内部迭代子（Internal Iterator）是主动和被动迭代子的等义词。这与本书所说的外禀迭代子（Extrinsic Iterator）和内禀迭

代子（Intrinsic Iterator）是不同的概念。

本书所指的内禀迭代子是定义在聚集结构内部的迭代子，而外禀迭代子是定义在聚集结构外部的迭代子，其他很多文献（如[ALPERT98]）中都有相同的提法。

Java 语言的设计师们在设计 AbstractList 类时，选择了使用内禀迭代子类，也即 Itr。但同时这个类也向外部提供自己的遍历方法，换言之，如果设计师使用 AbstractList 聚集，也同样可以定义自己的外禀迭代子。

那么在什么情况下选择内禀迭代子，什么情况下选择外禀迭代子呢？

一个外禀迭代子往往仅存储一个游标，因此如果有几个客户端同时进行迭代的话，那么可以使用几个外禀迭代子对象，由每一个迭代子对象控制一个独立的游标。但是，外禀迭代子要求聚集对象向外界提供遍历方法，因此会破坏对聚集的封装。如果某一个客户端可以修改聚集元素的话，迭代会给出不自恰的结果，甚至影响到系统其他部分的稳定性，造成系统崩溃。

使用外禀迭代子的一个重要理由是它可以被几个不同的方法和对象共同享用和控制。使用内禀迭代子的优点是它不破坏对聚集的封装。

静态迭代子和动态迭代子

静态迭代子由聚集对象创建，并持有聚集对象的一个快照（snapshot），在产生后这个快照的内容就不再变化。客户端可以继续修改原聚集的内容，但是迭代子对象不会反映出聚集的新变化。

静态迭代子的好处是它的安全性和简易性，换言之，静态迭代子易于实现，不容易出现错误。但是由于静态迭代子将原聚集复制了一份，因此它的短处是对时间和内存资源的消耗。对大型的聚集对象来说，使用静态迭代子不是一个合适的选择。

动态迭代子则与静态迭代子完全相反，在迭代子被产生之后，迭代子保持着对聚集元素的引用，因此，任何对原聚集内容的修改都会在迭代子对象上反映出来。

完整的动态迭代子不容易实现，但是简化的动态迭代子并不难实现。大多数 Java 设计师遇到的迭代子都是这种简化的动态迭代子。为了说明什么是简化的动态迭代子，首先需要介绍一个新的概念：Fail Fast。

Fail Fast

如果当一个算法开始之后，它的运算环境发生变化，使得算法无法进行必需的调整时，这个算法就应当立即发出故障信号。这就是 Fail Fast 的含义。

如果聚集对象的元素在一个动态迭代子的迭代过程中发生变化时，迭代过程会受到影响而变得不能自恰。这时候，迭代子就应当立即抛出一个异常。这种迭代子就是实现了 Fail Fast 功能的迭代子。

本章给出的例子并没有将 Fail Fast 功能放到设计中去，但是本书的“专题：Java 对迭代子模式的支持”一章对 Fail Fast 功能有进一步的讨论。



过滤迭代子

一个普通的迭代子仅仅扫过一个聚集对象的所有元素，一个过滤迭代子则不同。

过滤迭代子可以在扫过聚集元素的同时进行计算，以确定呈现给客户端的元素都是满足某一个过滤条件的，或者这些元素已经经过了重新的排列等。

43.4　迭代子模式的优点和缺点

迭代子模式有如下的优点：

（1）迭代子模式简化了聚集的界面。迭代子具备了一个遍历接口，这样聚集的接口就不必具备遍历接口。

（2）每一个聚集对象都可以有一个或一个以上的迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。因此，一个聚集对象可以同时有几个迭代在进行之中。

（3）由于遍历算法被封装在迭代子角色里面，因此迭代的算法可以独立于聚集角色变化。由于客户端拿到的是一个迭代子对象，因此，不必知道聚集对象的类型，就可以读取和遍历聚集对象。这样即便聚集对象的类型发生变化，也不会影响到客户端的遍历过程。

迭代子模式的缺点是：

（1）迭代子模式给客户端一个聚集被顺序化的错觉，因为大多数的情况下聚集的元素并没有确定的顺序，但是迭代必须以一定线性顺序进行。如果客户端误以为顺序是聚集本身具有的特性而过度依赖于聚集元素的顺序，很可能得出错误的结果。

（2）迭代子给出的聚集元素没有类型特征。一般而言，迭代子给出的元素都是 Object 类型，因此，客户端必须具备这些元素类型的知识才能使用这些元素。

43.5　一 个 例 子

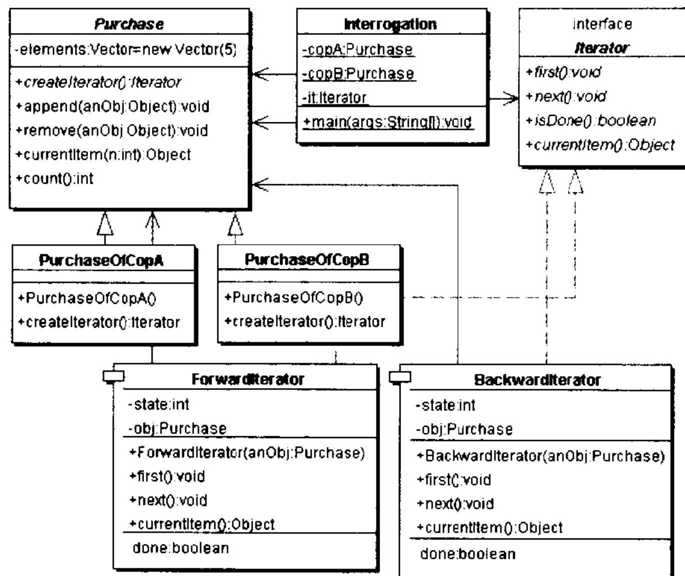
问题

有一天，两个纽约的警察审问一个涉嫌杀人的电器推销员。三个小时之后，他们去见他们的头儿说，他们不能再审下去了，因为他们已经从推销员那里买了一台微机、一台微波炉、两个吹风机。警察的审讯变成了嫌犯的产品推销会。头儿在震怒之余，搜出了他们购买的电器进行清点。

请设计一个系统模拟这个清点过程。为了增加清点的乐趣，请使用两个不同的清点方法：一个从前到后，一个从后到前。

设计

警察的审讯变成了嫌犯的推销会，现在清点他们购买的东西。下图所示就是这次清点的系统设计图。假想警察 A 和警察 B 购买的东西放在他们各人的购物筐中，那么购物筐就是包含有元素对象的容器对象，这样的容器对象就是聚集。



首先需要定义出一个抽象的购物筐，以便给出所有的具体购物筐所共有的行为，以及具体购物筐需要实现的方法，这就是抽象类 Purchase。两个警察每人的购物筐则由两个聚集类代表：PurchaseOfCopA 类和 PurchaseOfCopB 类，而它们继承于抽象类 Purchase。

聚集类由一个工厂方法返还一个迭代子对象。接口 Iterator 给出迭代所需的接口，而具体迭代子 ForwardIterator 和 BackwardIterator 分别是向前迭代的和向后迭代的具体迭代类。

通过封装一个 Vector 对象，抽象聚集类 Purchase 成为一个聚集，并且提供了必要的聚集管理方法。因为 Vector 类本身就是聚集，而 Purchase 类就不必从 Object[]这样的数组做起，而只需将 Vector 对象封装起来即可。

系统的源代码

抽象聚集类 Purchase 的源代码如代码清单 10 所示。

代码清单 10：Purchase 抽象类的源代码

```
import java.util.Vector;
public abstract class Purchase
{
    private Vector elements = new Vector(5);

  /**
```

```
     *  工厂方法：提供迭代子的实例
     */
     public abstract Iterator createIterator();
    /**
     *  聚集方法：将一个新元素增加到聚集的最后面
     */
     public void append(Object anObj)
     {
         elements.addElement(anObj);
     }
    /**
     *  聚集方法：删除一个元素
     */
     public void remove(Object anObj)
     {
         elements.removeElement(anObj);
     }
    /**
     *  聚集方法：返还一个元素
     */
     public Object currentItem(int n)
     {
             return elements.elementAt(n);
     }
    /**
     *  聚集方法：返还聚集的大小
     */
     public int count()
     {
         return elements.size();
     }
}
```

两个警察的购物筐是两个具体聚集类，全都继承自抽象聚集类。下面这个购物筐 PurchaseOfCopA 是属于警察 A 的，它的工厂方法 createIterator()提供一个正向迭代子的实例，如代码清单 11 所示。

代码清单 11：具体聚集类 PurchaseOfCopA 的源代码

```
public class PurchaseOfCopA extends Purchase
{
     public PurchaseOfCopA(){}
    /**
     *  工厂方法：返还一个正向迭代子
     */
     public Iterator createIterator()
     {
```

```
        return new ForwardIterator(this);
    }
}
```

下面是警察 B 的购物筐 PurchaseOfCopB 的源代码，它的工厂方法 createIterator()提供一个反向迭代子的实例，如代码清单 12 所示。

代码清单 12：具体聚集类 PurchaseOfCopB 的源代码

```
public class PurchaseOfCopB extends Purchase
{
    public PurchaseOfCopB(){}
   /**
    *  工厂方法：返还一个反向迭代子
    */
    public Iterator createIterator()
    {
        return new BackwardIterator(this);
    }
}
```

抽象迭代子角色由接口 Iterator 扮演，它声明了所有的具体迭代子所需的接口，包括 first()、next()、isDone()以及 currentItem()等方法，如代码清单 13 所示。

代码清单 13：接口 Iterator 的源代码

```
public interface Iterator
{
    /**
     *  迭代方法：移动到第一个元素
     */
    void first();
   /**
    *  迭代方法：移动到下一个元素
    */
    void next();
   /**
    *  迭代方法：是否是最后一个元素
    */
    boolean isDone();
   /**
    *  迭代方法：返还当前元素
    */
    Object currentItem();
}
```

具体迭代子类 ForwardIterator 实现了抽象迭代子角色所声明的接口。从下面的源代码可以看出，这是一个正向迭代的迭代子。它从第一个元素开始，到最后一个元素为止，进行正向迭代，如代码清单 14 所示。

代码清单 14：ForwardIterator 类的源代码

```
public class ForwardIterator implements Iterator
{
    private int state;
    private Purchase obj;
   /**
    *  构造子
    */
    public ForwardIterator(Purchase anObj)
    {
        obj = anObj;
    }
   /**
    *  迭代方法：移动到第一个元素
    */
    public void first()
    {
        state = 0;
    }
   /**
    *  迭代方法：移动到下一个元素
    */
    public void next()
    {
        if (!isDone())
        {
            state++;
        }
    }
   /**
    *  迭代方法：是否到达最后一个元素
    */
    public boolean isDone()
    {
        if (state>obj.count() - 1)
        {
            return true;
        }
        return false;
    }
   /**
    *  迭代方法：取出当前的元素
    */
    public Object currentItem()
    {
        return obj.currentItem(state);
```

```
        }
    }
```

具体迭代子类 BackwardIterator 实现了抽象迭代子角色所声明的接口。从下面的源代码可以看出，这是一个逆向迭代的迭代子。它从最后一个元素开始，到第一个元素为止，进行反向迭代，如代码清单 15 所示。

代码清单 15：向后迭代的源代码

```
public class BackwardIterator implements Iterator
{
    private int state;
    private Purchase obj;
    /**
     *  构造子
     */
    public BackwardIterator(Purchase anObj)
    {
        obj = anObj;
    }
    /**
     *  迭代方法：移动到最后的一个元素
     */
    public void first()
    {
        state = obj.count() - 1;
    }
    /**
     *  迭代方法：移动到前一个元素
     */
    public void next()
    {
        if (!isDone())
        {
            state--;
        }
    }
    /**
     *  迭代方法：是否到达第一个元素
     */
    public boolean isDone()
    {
        if (state
```

系统运行的结果给出两个警察所购买的电器产品。这里用向前迭代的方式给出第一个警察所购的物品，用向后迭代的方式给出第二个警察所购的物品。

运行的结果如代码清单 16 所示。

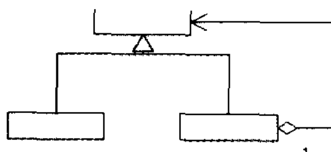
代码清单 16：系统的运行结果

```
Creating forward iterator:
Item: No. 1: Dish Washer
Item: No. 2: Hair Dresser
Item: No. 3: Microwave
Creating backward iterator:
Item : No. 5: Dish Washer
Item : No. 4: PC
Item : No. 3: Digital Camera
Item : No. 2: Diskman
Item : No. 1: Hair Dresser
```

43.6 迭代子模式与其他模式的关系

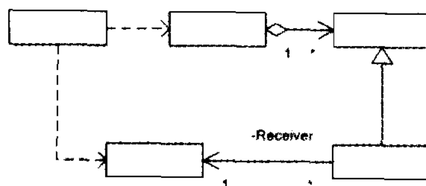
合成模式

迭代子模式常常用来遍历一个按照合成模式建立起来的树结构，如下图所示。



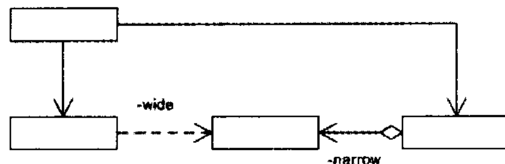
命令模式

命令模式常常需要维护一个历史清单聚集，并使用迭代子模式遍历历史清单的元素，命令模式需要这一功能来实现命令的撤销和恢复。命令模式的简略类图如下图所示。



备忘录模式

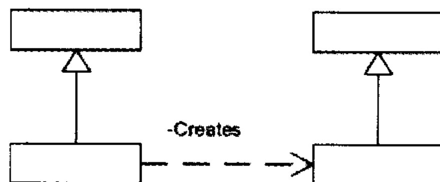
迭代子对象在迭代过程中会存储迭代的状态，包括游标的位置等，这是备忘录模式的应用。备忘录模式的简略类图如下图所示。



如果考察一下外禀迭代子对象就会发现，外禀迭代子实际上就是一个存储了迭代游标状态的备忘录对象。

工厂方法模式

聚集角色一般提供一个工厂方法，以向外界提供自己的迭代子对象。在上面给出的示意性实现中，抽象聚集角色 Aggregate 和具体聚集角色 ConcreteAggregate 的 createIterator() 方法就是工厂模式的典型应用。工厂方法模式的简略类图如下图所示。



问答题

1. 在上面给出的黑箱聚集的迭代子模式的示意性实现里，具体聚集对象封装了一个数组作为聚集元素的容器，这是否意味着迭代子模式要求设计师自行设计新的聚集？

2. 请设计一个（动态）迭代子类，其构造子接收一个 Vector 聚集对象作为参量。这个迭代子提供 Vector 聚集的迭代功能。

3. 请改造上面的动态迭代子类为静态迭代子类。

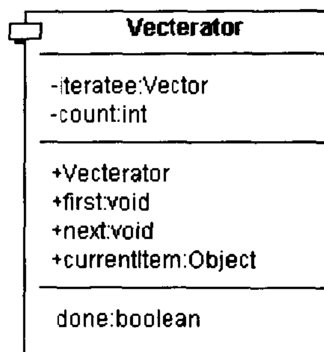
问答题答案

1. 在本章前面给出的迭代子模式的示意性实现里面，具体聚集对象封装了一个数组作为聚集。但是由于 Java 语言本身提供了许多的聚集对象，因此读者完全可以使用这些聚集，而不必从数组开始。比如读者完全可以从一个 Vector 对象开始设计一个具体聚集对象。

事实上，本书推荐读者尽量利用 Java 语言已经提供的功能，而迭代子模式并没有建议设计师自行设计新的聚集。

设计模式并不能“要求”设计师做什么。设计模式是一些设计的范例，供设计师进行学习和交流。一个具体系统的设计必须从该系统本身的要求和特点出发进行设计，不可生搬硬套，非要将模式照搬到自己的设计中去；或者削足适履，修改自己的系统设计以符合设计模式的描述。

2. 本书将英文字 Vector 和 Iterator 合并起来，造出 Vecterator 这个词作为这个类的名字。Vecterator 的设计图如右图所示。



可以看出，Vecterator 类提供了 first()、next()、currentItem()、isDone()等迭代方法，它的构造子接收一个 Vector 对象作为参量。

这个类的源代码如代码清单 17 所示。

代码清单 17：动态迭代子 Vecterator 的源代码

```
package com.javapatterns.iterator.vecterator;
import java.util.Vector;
import java.util.NoSuchElementException;
public class Vecterator
{
    private Vector iteratee;
    private int count;
  /**
   *  构造子
   */
    public Vecterator(Vector v)
    {
        iteratee = v;
        count = 0;
    }
  /**
   *  迭代方法：移动到第一个元素
   */
    public void first()
    {
        count = 0;
    }
  /**
   *  迭代方法：是否是最后一个元素
   */
    public boolean isDone()
    {
        return count < iteratee.size();
    }
  /**
   *  迭代方法：移动到下一个元素
   */
    public void next()
    {
        if (count < iteratee.size())
        {
```

```
                count++;
            }
        }
    /**
     *  迭代方法：返还当前元素
     */
      public Object currentItem()
      {
            synchronized (iteratee)
            {
                  if (count < iteratee.size())
                  {
                        return iteratee.elementAt(count);
                  }
            }
            throw new NoSuchElementException("Vecterator");
      }
}
```

3. 新的静态迭代子 Vecterator 的设计图与上题中动态迭代子的设计图没有区别，但是源代码不同，如代码清单 18 所示。

代码清单 18：静态迭代子 Vecterator 的源代码

```
package com.javapatterns.iterator.vecterator2;
import java.util.Vector;
import java.util.NoSuchElementException;
public class Vecterator
{
      private Vector iteratee;
      private int count;
    /**
     *  构造子
     */
      public Vecterator(Vector v)
      {
            iteratee = (Vector) v.clone();
            count = 0;
      }
    /**
     *  迭代方法：移动到第一个元素
     */
      public void first()
      {
          count = 0;
      }
    /**
     *  迭代方法：是否是最后一个元素
```



```
    */
    public boolean isDone()
    {
        return count < iteratee.size();
    }
    /**
     *  迭代方法：移动到下一个元素
     */
    public void next()
    {
        if (count < iteratee.size())
        {
            count++;
        }
    }
    /**
     *  迭代方法：返还当前元素
     */
    public Object currentItem()
    {
        synchronized (iteratee)
        {
            if (count < iteratee.size())
            {
                return iteratee.elementAt(count);
            }
        }
        throw new NoSuchElementException("Vecterator");
    }
}
```

进行比较后可以发现，静态迭代子 Vecterator 与动态迭代子 Vecterator 的惟一区别在于静态迭代子的构造子将传入的 Vector 对象复制了一份，从而持有一份与原来聚集完全独立的拷贝，而动态迭代子持有一个指向原来的聚集的引用。静态迭代子 Vecterator 源代码如代码清单 19 所示。

代码清单 19：静态迭代子 Vecterator 的源代码

```
    iteratee = v.clone();
```

第 44 章　专题：Java 对迭代子模式的支持

在阅读本章之前，请先阅读本书的“迭代子（Iterator）模式”一章。

44.1　Java 聚集

Java 对迭代子模式的支持来自于 Java 聚集的需要：一个聚集需要向外界提供遍历聚集元素的方法。

Java 聚集是一个庞大的家族，但是如果认真考察就会发现，Java 聚集大体上可以分成三种：Map，List 和 Set，而每一种都只有非常有限的几种实现而已。Java 聚集的共同之处是它们都实现了 java.util.Collection 接口，这个接口的源代码如代码清单 1 所示。

代码清单 1：Collection 接口的源代码

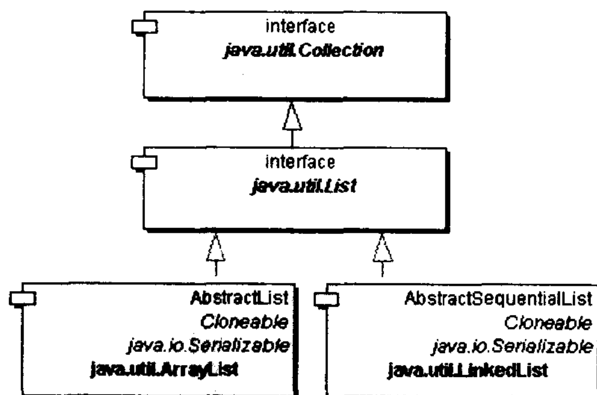
```
package java.util;
public interface Collection
{
    // 查询操作
   /**
    *  返还聚集的大小
    */
    int size();
   /**
    *  聚集是否为空
    */
    boolean isEmpty();
    boolean contains(Object o);
   /**
    *  工厂方法，返还迭代子对象
    */
    Iterator iterator();
    Object[] toArray();
    Object[] toArray(Object a[]);
    // 修改操作
   /**
    *  增加一个聚集元素
    */
```

```
        boolean add(Object o);
       /**
        *  删除一个聚集元素
        */
        boolean remove(Object o);
        // 多项操作
        boolean containsAll(Collection c);
        boolean addAll(Collection c);
        boolean removeAll(Collection c);
        boolean retainAll(Collection c);
        void clear();
        boolean equals(Object o);
        int hashCode();
    }
```

List 聚集

作为 Collection 接口的子接口，List 接口规定出标准的 List 聚集必须实现的方法。所有实现了 List 接口的 Java 聚集都称做 List 聚集。

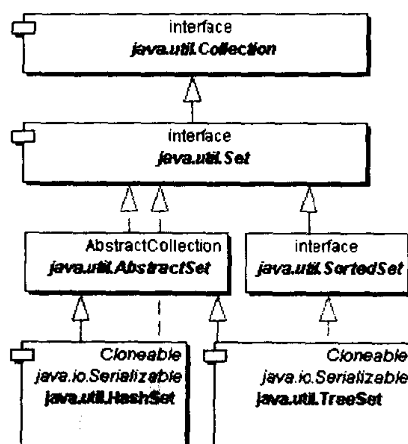
对于一个 List 聚集对象来说，聚集元素的顺序是很重要的。因此，List 提供了几种方法在列的中间删除或者增加元素。如下图所示，其中给出了两种最为常见的 List 聚集，以及它们所实现的接口。



Set 聚集

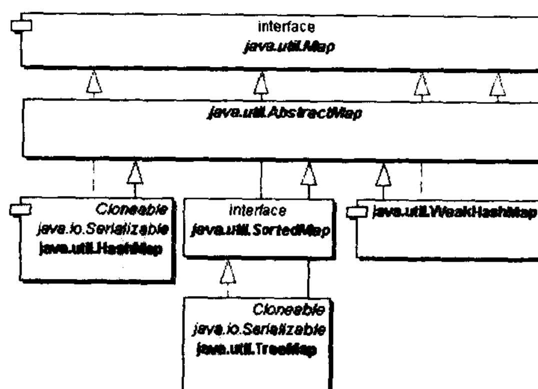
所有实现了 java.util.Set 接口的 Java 聚集都成为 Set 聚集。Set 接口与 Collection 接口并无区别，但是 Set 的行为有特殊之处。Set 聚集中的元素必须是惟一的，因此，每个 Set 聚集对象都提供衡量两个元素是否相同的方法。

Set 聚集的一个显著特点是不保持元素的顺序。如下图所示，其中给出了两个最为常见的 Set 聚集，以及它们所实现的接口。



Map 聚集

实现了 java.util.Map 接口的 Java 聚集称为 Map 聚集。Map 聚集存储键-元素对（key-object pair），允许根据一个元素的键查找元素。Java 提供两种基本 Map 聚集，即 HashMap 和 TreeMap，如下图所示。

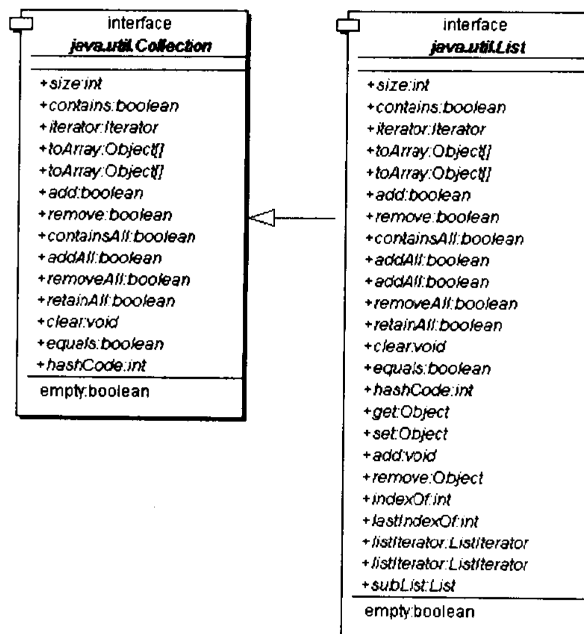


44.2　Java 聚集中的迭代子

正如上面所谈到的，一般的聚集对象都应当提供某种迭代功能，以允许客户端遍历所有的聚集元素。Java 聚集使用迭代子模式提供迭代功能。

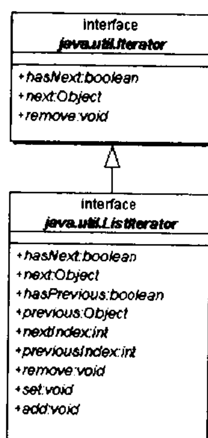
Java 聚集接口

Java 的聚集接口 java.util.Collection 是 java.util 库中所有的聚集类都要实现的接口。作为 Collection 类型的子类型，java.util.List 是 java.util 库中所有的 List 类型都必须实现的接口。Collection 和 List 是 Java 聚集中最重要的接口，如下图所示，其中就显示了这两种接口的关系。



Iterator 与 ListIterator 接口

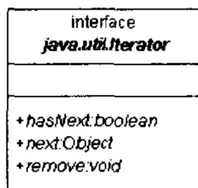
Collection 接口有一个方法 iterator()，返还一个 java.util.Iterator 类型；而 java.util.List 接口的 listIterator()方法返还一个 java.util.ListIterator 类型。这两个类型构成 Java 语言对迭代子模式的支持。Iterator 接口和 ListIterator 接口的静态关系图如下图所示。



44.3 java.util.Iterator 接口

Java 语言的 java.util.Iterator 接口就是这样一个迭代子模式的应用。Iterator 接口的类图如下图所示。





首先看一看这个接口的源代码，如代码清单 2 所示。

代码清单 2：java.util.Iterator 接口的源代码

```
package java.util;
public interface Iterator
{
    /**
     * 返还 true，如果前面还有元素
     */
    boolean hasNext();
    /**
     * 返还迭代中的下一个元素
     */
    Object next();
    /**
     * 删除最后遍历过的元素
     */
    void remove();
}
```

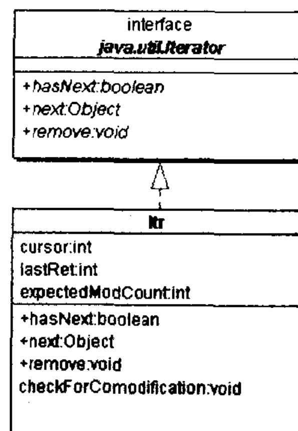
通过调用聚集类的 iterator()方法，可以得到一个 Iterator 类型的对象，而一个 Iterator 类型的对象有以下的这些功能：

（1）通过调用 next()方法可以得到序列上的下一个对象。对象在初始状态时调用 next()，便返回序列上的第一个元素。

（2）通过调用 hasNext()可以查看序列的后面还有没有元素。

（3）调用 remove()可以把序列上当前的元素删除掉，这个删除方法是可选的。如果不支持这个方法，在调用时可以抛出 java.lang.UnsupportedOperationException 异常。

怎样实现 Iterator 接口

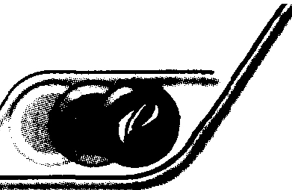


既然 Java 语言以接口 java.util.Iterator 的方式支持迭代子模式，那么不妨先看一看 Java 语言是怎么实现这个接口的。前面说过，Collection 接口要求提供 iterator()方法，此方法在调用时返还一个 Iterator 类型的对象。而作为 Collection 接口的子类型，AbstractList 类的内部成员类 Itr 便是实现 Iterator 接口的类，如右图所示。

Itr 类的源代码如代码清单 3 所示。

代码清单 3：java.util.AbstractList 类的内部类 Itr 的源代码

```
import java.util.Iterator;
class Itr implements Iterator
{
    /**
     * 再次调用 next()方法时所用的指标
     */
    int cursor = 0;
    /**
     * 最近一次调用时所用的指标
     */
    int lastRet = -1;
    int expectedModCount = modCount;
    /**
     * 返还 true，如果前面还有元素
     */
    public boolean hasNext()
    {
        return cursor != size();
    }
    /**
     * 返还迭代中的下一个元素
     */
    public Object next()
    {
        try
        {
            Object next = get(cursor);
            checkForComodification();
            lastRet = cursor++;
            return next;
        }
        catch(IndexOutOfBoundsException e)
        {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }
    /**
     * 删除最后遍历过的元素
     */
    public void remove()
    {
        if (lastRet == -1)
            throw new IllegalStateException();
        checkForComodification();
```



```
            try
            {
                AbstractList.this.remove(lastRet);
                if (lastRet < cursor)
                    cursor--;
                lastRet = -1;
                expectedModCount = modCount;
            }
            catch(IndexOutOfBoundsException e)
            {
                throw new ConcurrentModificationException();
            }
        }
        /**
         * 检查是否刚刚被另一个客户端修改过
         */
        final void checkForComodification()
        {
            if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        }
    }
```

仍以 AbstractList 类为例，它的 iterator()方法返还一个 Itr 类的实例，而 Itr 是 Iterator 类型的类。但是，AbstractList 类也有另一个方法 listIterator()，返还一个 ListItr 类的实例。

Fail Fast

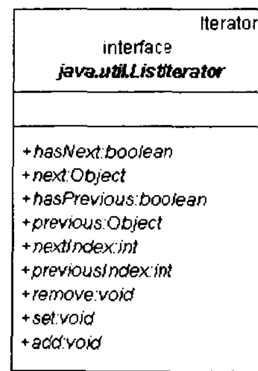
如同本书在“迭代子（Iterator）模式”一章中所讨论的，Fail Fast 是一种出现错误时的早期报警功能。一个动态迭代子的迭代是建立在原聚集对象的基础之上的，如果迭代子所遍历的聚集对象被外部修改，就会造成迭代过程崩溃。这时候，迭代子应当在崩溃可能发生之前尽早抛出一个异常。

从 Itr 类的源代码中可以看到，方法 checkForComodification()会检查聚集的内容是否刚刚被外界直接修改过（不是通过迭代子提供的 remove()方法修改的）。如果在迭代开始后，聚集的内容被外界绕过迭代子对象而直接修改过的话，这个方法会立即抛出 ConcurrentModificationException 异常。

这就是说，AbstractList.Itr 迭代子是一个 Fail Fast 的迭代子。

44.4　Java 迭代子接口 ListIterator

接口 java.util.ListIterator 是接口 java.util.Iterator 的扩展接口，可以与有序的聚集或者列（list）一同使用。ListIterator 接口的类图如下图所示。



从上面的类图可以看出，这个迭代子不仅提供了正向迭代（forward iterate）和逆向迭代（backward iterate）的方法，而且为可以变化的列提供了在迭代过程中安全地插入新的元素、删除已有的元素和更新已有元素的方法。

下面就是这个接口的源代码，如代码清单 4 所示。

代码清单 4：java.util.ListIterator 接口的源代码

```
package java.util;
public interface ListIterator extends Iterator
{
    /**
     * 返还 true，如果前面还有元素
     */
    boolean hasNext();
    /**
     * 返还迭代中的下一个元素
     */
    Object next();
    /**
     * 返还 true，如果后面还有元素
     */
    boolean hasPrevious();
    /**
     * 返还迭代中的上一个元素
     */
    Object previous();
    int nextIndex();
    int previousIndex();
    // Modification Operations
    /**
     * 删除当前的元素
     */
    void remove();
    void set(Object o);
    void add(Object o);
}
```



正向迭代

与 java.util.Iterator 一样，java.util.ListIterator 接口中最为常用的方法仍然是 hasNext()和 next()，它们对一个列做正向迭代。其中，next()方法返还列的下一个元素。如果列中没有下一个元素，就抛出 NoSuchElementException 异常。

如果此时调用 next()能够返还一个元素的话，则 hasNext()方法返还 true，反之，就返还 false。

逆向迭代

除了这两个方法之外，ListIterator 还定义了 hasPrevious()和 previous()两个方法，允许程序能够对一个列做逆向迭代。其中，previous()方法返还列的上一个元素。如果列中没有上一个元素，就抛出 NoSuchElementException 异常。

如果此时调用 previous()能够返还一个元素的话，hasPrevious()方法返还 true，反之，就返还 false。

此外，ListIterator 的 nextIndex()给出 next()会返还的元素的位置，而 previousIndex()给出 previous()会返还的元素的位置。

如果 next()抛出一个 NoSuchElementException 异常的话，nextIndex()会返还列的大小；如果 previous()抛出一个 NoSuchElementException 异常的话，previousIndex()就返还-1。

安全的修改方法

java.util.ListIterator 还声明了三个功能强大的方法，允许**在迭代的进行过程当中**安全地修改列的内容。这三个方法是一个实现 ListIterator 接口的类可以选择实现或者选择不实现的。

add()方法

add()方法将一个新的对象插入到列的当前位置上。严格地讲，就是在 next()方法返还的对象的后面。这就意味着，在调用 add()方法增加了一个新的对象后，next()方法并不会返回这个元素；相反，previous()方法会返回这个元素。

remove()方法

remove()方法将当前的元素删除。然而何谓“当前”？如果程序刚刚调用了 next()的话，删除的就是 next()方法返还过的那个元素；如果程序刚刚调用了 previous()方法的话，删除的就是 previous()方法返还过的那个元素。



只能在调用一次 next()或者 previous()之后，调用一次 remove()。如果需要再次调用 remove()的话，就需要再调用一次 next()或者 previous()，然后才可以调用 remove()。

如果刚刚调用了 add()，那么只有在调用 next()或者 previous()一次之后，才可以调用

remove()。

set()方法

set()方法会将当前的元素更新掉。所谓当前，也是和删除方法相同：如果程序刚刚调用了 next()的话，置换掉的就是 next()返还过的那个元素；如果程序刚刚调用了 previous()方法的话，置换掉的就是 previous()返还过的那个元素。

同样，只能在调用一次 next()或者 previous()之后，调用 set()一次。如果需要再次调用 set()的话，就需要再调用一次 next()或者 previous()，然后才可以调用 set()。

如果刚刚调用了 add()或者 remove()，那么只有在调用一次 next()或者 previous()之后，才可以调用 set()。

如果一个列是一个不变列，那么这三个修改方法也就不必实现；相反，如果一个列是可变列的话，这三个修改方法就必须实现。一个不支持这三个修改方法的列可以在客户端错误地调用这三个方法之一时，抛出 java.lang.UnsupportedOperationException 异常。

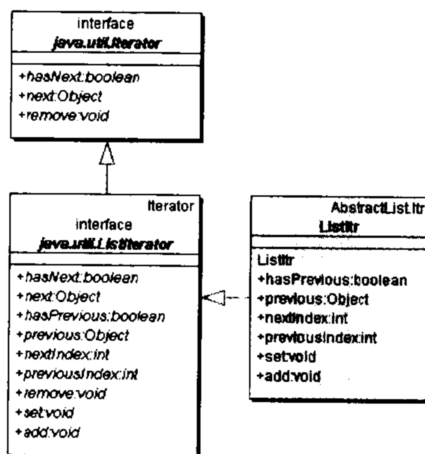
可变聚集和不变聚集

一个内部元素不会改变的聚集叫做不变聚集，反之就叫做可变聚集。（关于不变模式的讨论请参见本书的“不变（Immutable）模式”一章。）

如果一个 List 聚集是可变聚集的话，这三个方法就是应当被实现的。而相应地，所有对列的修改都应当通过 ListIterator 对象，而不是直接针对列本身。如果迭代正在进行，而列本身被直接修改，那么 ListIterator 会抛出一个 ConcurrentModificationException 异常。

怎样实现 ListIterator 接口

Java 语言中的 AbstractList、AbstractSequentialList、LinkedList 以及 List 等列提供的 listIterator()方法会返还一个 ListIterator 对象。仍以 AbstractList 为例，这个类通过提供一个内部子类 ListItr 实现了 ListIterator 接口。当 listIterator()方法被调用时，会返还这个内部类的实例。AbstractList 的内部成员类 ListItr 实现了 ListIterator 接口，如下图所示。



下面给出这个内部类的源代码，如代码清单 5 所示。读者可以看出，这个内部类实现

了向两个方向的迭代功能，以及增加新的聚集元素、更新已有的聚集元素的功能。这个 LitItr 类是 Itr 的子类，这意味着它会自动得到 Itr 的元素删除功能。

代码清单 5：内部类 ListItr 的源代码

```
private class ListItr extends Itr
    implements ListIterator
{
    ListItr(int index)
    {
        cursor = index;
    }
    public boolean hasPrevious()
    {
        return cursor != 0;
    }
    public Object previous()
    {
        try
        {
            Object previous = get(--cursor);
            checkForComodification();
            lastRet = cursor;
            return previous;
        }
        catch(IndexOutOfBoundsException e)
        {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }
    public int nextIndex()
    {
        return cursor;
    }
    public int previousIndex()
    {
        return cursor-1;
    }
    public void set(Object o)
    {
        if (lastRet == -1)
        throw new IllegalStateException();
            checkForComodification();
        try
        {
            AbstractList.this.set(lastRet, o);
```

```
            expectedModCount = modCount;
        }
        catch(IndexOutOfBoundsException e)
        {
            throw new ConcurrentModificationException();
        }
    }
    public void add(Object o)
    {
        checkForComodification();
        try
        {
            AbstractList.this.add(cursor++, o);
            lastRet = -1;
            expectedModCount = modCount;
        }
        catch(IndexOutOfBoundsException e)
        {
            throw new ConcurrentModificationException();
        }
    }
}
```

由于 ListItr 扩展了 Itr 类，因此，ListItr 会自动继承到 Itr 类的 next()方法和 remove()方法。

Fail Fast

尽管 AbstractList.ListItr 迭代子类允许客户端在迭代过程当中通过迭代子对象增加和删除聚集元素，但是不通过迭代子就直接对聚集元素进行修改仍然会造成迭代失败。为避免迭代因外界的直接修改而崩溃，AbstractList.ListItr 迭代子必须实现 Fail Fast 功能。

从 ListItr 类的源代码中可以看到，在 previous()方法、set()方法和 add()方法中，均调用 checkForComodification()方法检查聚集的内容是否刚刚被外界修改过。如果在迭代开始后聚集的内容被外界直接修改过（不是通过迭代子自己提供的 add()和 remove()等方法）的话，会立即抛出 ConcurrentModificationException 异常。

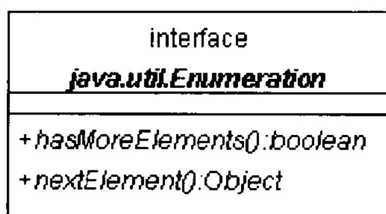
这就是说，AbstractList.ListItr 迭代子是一个具有 Fail Fast 功能的迭代子。

44.5 Enumeration 与 Iterator

Enumeration 接口

读到这里，熟悉 Java 语言特别是早期版本的 Java 语言的读者，可能会想到 Java 语言

的 API 中常常用到的 Enumeration 类型，如下图所示。



Enumeration 是 Java 1.2 版本之前使用的迭代接口。Java 1.2 版本提出了全新的 Java 聚集概念，与这个概念同时给出的是 Iterator 和 ListIterator 接口。

虽然 Java 的文档无不鼓励设计师使用 Iterator，而不要再使用 Enumeration，但是已有的各种在 Java 1.2 版本之前设计的系统仍然提供 Enumeration 类型。

那么一个自然的问题就是，Enumeration 与 Iterator 以及 ListIterator 这三种迭代接口以及实现到底有什么样的区别呢？

在接口上的区别

从接口上来看，Enumeration 没有 remove()方法，而 Iterator 接口声明了这一方法。remove()方法允许客户端能够在迭代过程当中将刚刚读出的聚集元素删除掉。

与 Enumeration 相比较，ListIterator 接口则给出了更多的方法。后者的额外功能主要体现在两个方面：

（1）对反向迭代的支持。ListIterator 接口所提供的方法允许客户端在两个方向上迭代，包括正向和反向。

（2）对增加聚集元素和删除聚集元素的支持。

显然，Enumeration 给出的是一个最小化的迭代子接口，而 Iterator 与 ListIterator 接口都比它大。

单纯从接口来看，以 Iterator 与 Enumeration 最为接近，实际上区别并不大。考虑到有些具体迭代子实际上并不实现 remove()方法，两者在接口上看更是等效的。本书在“适配器（Adapter）模式”一章中曾给出两个例子，说明如何使用适配器模式在 Enumeration 和 Iterator 之间进行转换。

在实现上的区别

java.util.Vector 是 Java 1.2 版本之前就提供的，在 Java 1.2 给出聚集框架时，Vector 成为聚集框架中的一员。为了提供对老系统的支持，Vector 同时支持 Enumeration 和 Iterator 接口。也就是说，在 Vector 的源代码中可以找到 Enumeration 的实现。

下面就是 Vector 对象的 elements()方法的源代码，如代码清单 6 所示。

代码清单 6：Vector 类的源代码片断

```
public Enumeration elements() {
return new Enumeration() {
```

```
        int count = 0;
        public boolean hasMoreElements() {
        return count < elementCount;
        }
        public Object nextElement() {
        synchronized (Vector.this) {
            if (count < elementCount) {
            return elementData[count++];
            }
        }
        throw new NoSuchElementException("Vector Enumeration");
        }
    };
    }
```

可以看出，这个方法返还一个 Enumeration 类型的对象，这个对象是 Vector 的一个内部无名类的实例。如果读者将这段源代码与前面所给出的 Itr 和 ListItr 等内部类的源代码相比较，就会发现一个重要的不同之处：Enumeration 不支持 Fail Fast。

Enumeration 不支持 Fail Fast

换言之，假设一个迭代过程是使用 Enumeration 进行的，那么如果在迭代过程中，聚集对象被外界意外直接修改（不是通过迭代子自己提供的修改方法进行），则这个迭代过程不会立即捕获任何异常。但是，由于聚集内容已经被修改，所以迭代很可能会在最后遇到不能自恰的内容时崩溃。

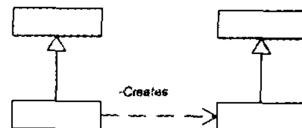
由于这个原因，本书建议读者尽量不要使用 Enumeration 对象，应当尽量使用 Iterator 对象。

44.6 其他相关模式

工厂方法模式

Collection 接口的 iterator()方法和 java.util.List 接口的 listIterator()方法都是工厂方法的典型应用。

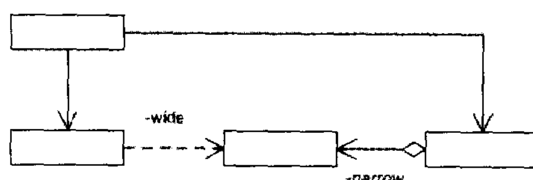
当客户端调用一个聚集的 iterator()方法时，此方法会返还其内部某个迭代类的一个实例。但是这里面的细节客户端不会知道，客户端所知道的仅仅是返还类型是 Iterator 类型。工厂方法模式的简略类图如右图所示。



每一个聚集对象都可以有多个工厂方法，它们返还不同的迭代子对象。

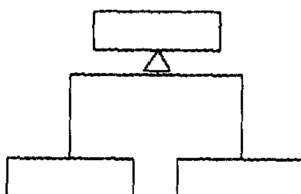
备忘录模式

AbstractList 含有 Itr 子类和 ListItr 子类，AbstractList 相当于备忘录模式中的发起人和负责人。而 Itr 子类和 ListItr 子类相当于备忘录角色，它们所存储起来的迭代游标就是备忘录对象所存储的状态。客户端根据备忘录所存储的状态信息，决定下一步迭代的游标位置。备忘录模式的简略类图如下图所示。



原始模型模式

所有的 Java 聚集对象都实现了 Cloneable 接口，并实现了 clone()方法，这是原始模型模式的应用。原始模型模式的简略类图如下图所示。



第 45 章　责任链（Chain of Responsibility）模式

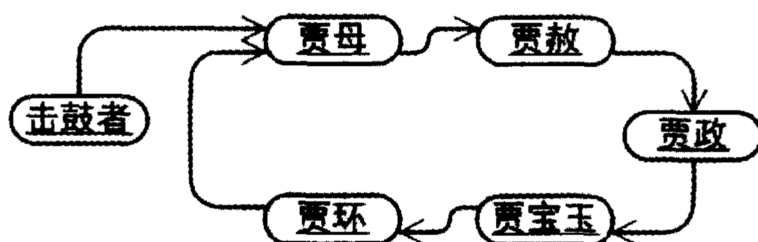
责任链模式是一种对象的行为模式[GOF95]。

在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

45.1　从击鼓传花谈起

击鼓传花是一种热闹而又紧张的饮酒游戏。在酒宴上宾客依次坐定位置，由一人击鼓，击鼓的地方与传花的地方是分开的，以示公正。开始击鼓时，花束就开始依次传递，鼓声一落，如果花束在某人手中，则该人就得饮酒。

比如说，贾母、贾赦、贾政、贾宝玉和贾环是五个参加击鼓传花游戏的传花者，他们组成一个环链。击鼓者将花传给贾母，开始传花游戏。花由贾母传给贾赦，由贾赦传给贾政，由贾政传给贾宝玉，又由贾宝玉传给贾环，由贾环传回给贾母，如此往复，如下图所示。当鼓声停止时，手中有花的人就得执行酒令。

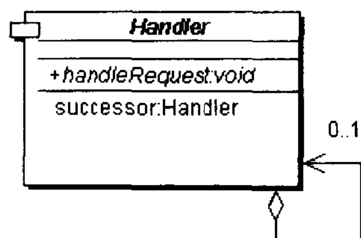


击鼓传花便是责任链模式的应用。责任链可能是一条直线、一个环链或者一个树结构的一部分。

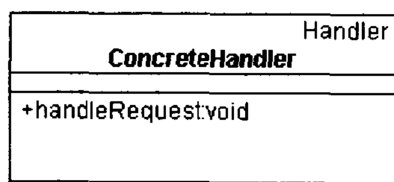
45.2　责任链模式的结构

为了讲解的方便，下面使用了一个责任链模式的最简单的实现。责任链模式涉及到的角色如下所示：

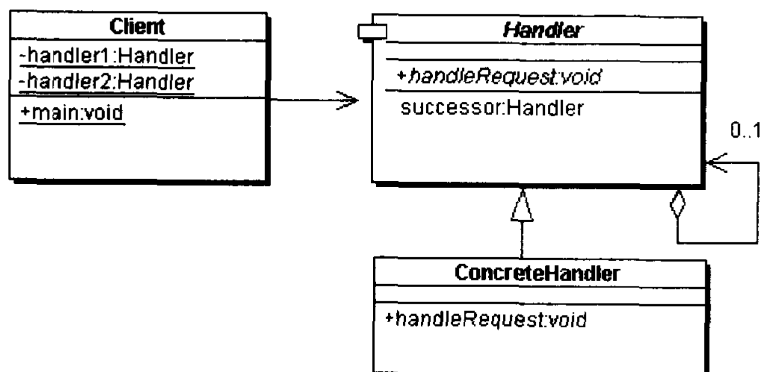
- 抽象处理者（Handler）角色：定义出一个处理请求的接口。如果需要，接口可以定义出一个方法，以设定和返回对下家的引用。这个角色通常由一个 Java 抽象类或者 Java 接口实现。其示意性类图如右图所示。图中的聚合关系给出了具体子类对下家的引用，抽象方法 handleRequest()规范了子类处理请求的操作。



- 具体处理者（ConcreteHandler）角色：具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。其示意性的类图如下图所示。



上图中的示意性的具体处理者 ConcreteHandler 类只有 handleRequest()一个方法。责任链模式的静态类结构如下图所示。

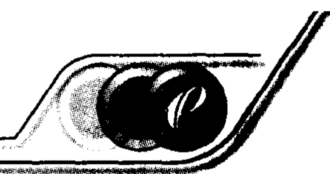


图中给出了一个客户端，以便读者可以更清楚地看到责任链模式是怎样应用的。

抽象处理者的示意性源代码如代码清单 1 所示。

代码清单 1：抽象处理者的源代码

```
package com.javapatterns.chainofresp;
public abstract class Handler
{
    /**
     *    处理方法，调用此方法处理请求
     */
    protected Handler successor;
    public abstract void handleRequest();
    /**
     *    赋值方法，调用此方法设定下家
```



```
     */
    public void setSuccessor(Handler successor)
    {
        this.successor = successor;
    }
    /**
     *    取值方法
     */
     public Handler getSuccessor()
    {
        return successor;
    }
}
```

下面是具体处理者的示意性源代码，如代码清单 2 所示。显然，处理者的逻辑非常简单，如果一个具体处理角色有下家，就将请求传递给下家；如果没有下家，就处理请求。

代码清单 2：具体处理者的源代码

```
package com.javapatterns.chainofresp;
public class ConcreteHandler extends Handler
{
    /**
     *    处理方法，调用此方法处理请求
     */
    public void handleRequest()
    {
        if (getSuccessor() != null)
        {
            System.out.println("The request is passed to "
                + getSuccessor());
            getSuccessor().handleRequest();
        }
        else
        {
            System.out.println("The request is handled here.");
        }
    }
}
```

当然在大多数情况下，这么简单的处理逻辑是没有实际用途的。真实的处理逻辑是与所研究的系统的商业逻辑密切相关的，这里使用最简化的商业逻辑，有助于读者将精力集中到如何将模式应用到设计中去。

客户端的源代码如代码清单 3 所示。

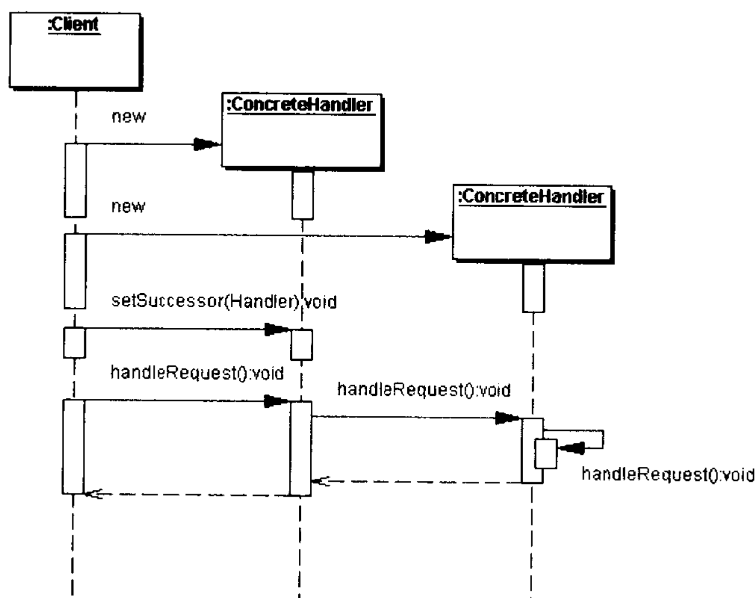
代码清单 3：客户端的源代码

```
package com.javapatterns.chainofresp;
```

```
public class Client
{
    static private Handler handler1, handler2;
    public static void main(String[] args)
    {
        handler1 = new ConcreteHandler();
        handler2 = new ConcreteHandler();
        handler1.setSuccessor(handler2);
        handler1.handleRequest();
    }
}
```

可以看出，客户端创建了两个处理者对象，并指定第一个处理者对象的下家是第二个处理者对象，而第二个处理者对象没有下家。然后客户端将请求传递给第一个处理者对象。

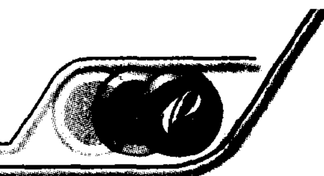
由于本系统的传递逻辑非常简单：只要有下家，就传给下家处理；如果没有下家，就自行处理。因此，第一个处理者对象接到请求后，会将请求传递给第二个处理者对象。由于第二个处理者对象没有下家，于是自行处理请求。责任链模式的示意性实现的活动时序图如下图所示。



45.3 纯的与不纯的责任链模式

一个纯的责任链模式要求一个具体的处理者对象只能在两个行为中选择一个：一是承担责任，二是把责任推给下家。不允许出现某一个具体处理者对象在承担了一部分责任后又把责任向下传的情况。

在一个纯的责任链模式里面，一个请求必须被某一个处理者对象所接收；在一个不纯的责任链模式里面，一个请求可以最终不被任何接收端对象所接收。



纯的责任链模式的实际例子很难找到，一般看到的例子均是不纯的责任链模式的实现。有些人认为不纯的责任链根本不是责任链模式，这也许是有道理的。但是在实际的系统里，纯的责任链很难找到。如果坚持责任链不纯便不是责任链模式，那么责任链模式便不会有太大的意义了。

45.4　Java 1.0 版的 AWT 事件处理机制

Java 1.0 版本中的 AWT 库使用了责任链模式和命令模式来处理 GUI 的事件。由于视窗构件往往位于容器构件内，因此，当事件发生在一个构件上时，此构件的事件处理器可以处理此事件，然后决定是否将事件向上级容器构件传播。上级容器构件接到事件后，可以在此处理此事件，然后决定是否将事件再次向上级容器构件传播，如此往复，直到事件到达顶层构件。

事件浮升机制

比如，当一个视窗构件接到一个 Mouse_Click 事件时，事件首先传播到它所发生的构件上，然后向其容器构件传播。容器可以选择处理这个事件，或者再将此事件向更高一级的容器构件传播。事件如此一级级地向上传播，就像水底的气泡一点一点地冒到水面上一样，因此又叫做事件浮升（Event Bubbling）机制。下面就是一段典型的 Java 1.0 版本的 AWT 库里处理事件的代码，如代码清单 4 所示。

代码清单 4：Java 1.0 版本中 AWT 处理事件的典型源代码

```
public boolean action(Event event, Object obj)
{
    if (event.target == btnOK)
    {
        doOKBtnAction();
    }
    else if (event.target == btnExit)
    {
        doExitBtnAction();
    }
    else
    {
        return super.action(event, obj);
    }
    return true;
}
```

在这段代码里，action()判断目标构件是不是 btnOK 或 btnExit。如果是，便运行相应的方法；如果不是，便返还 true，一个方法返还 true 便使得事件停止浮升。

AWT 1.0 事件处理模型的缺点

AWT 1.0 事件处理模型是基于继承的。为了使一个程序能够捕捉 GUI 的事件并处理此事件，必须继承此构件并且给其子类配备事件处理器，也就是置换掉 action()方法或者 handleEvent()方法。这不是应当提倡的做法：在一个面向对象的系统里，经常使用的应当是委派关系，继承关系不应当被滥用。

（1）在一个复杂的 GUI 系统里，这样为所有事件的构件提供子类，会导致很多的子类，增加维护成本。

当然，由于事件浮升机制，可以在构件的树结构的根（root）部件里面处理所有的事件。但是这样一来，就需要使用复杂的条件转移语句在这个根部部件里辨别事件的起源和处理方法。这种非常过程化的处理方法很难维护，并且与面向对象的设计思想相违背。

（2）由于每一个事件都会沿着构件树结构向上传播，因此事件浮升机制会使得事件的处理变得较慢，这也是缺点之一。

比如在有些操作系统中，鼠标每移动一个色素，都会激发一个 Mouse_Move 事件。每一个这样的事件都会沿着构件的容器树结构向上传播，这会使得鼠标事件成灾。

（3）AWT 1.0 的事件处理模型只适用于 AWT 构件类，这是此模型的另一个缺点。

责任链模式要求链上所有的对象都继承自一个共同的父类，这个类便是 java.awt.Component 类。

显然，由于每一级的构件在接到事件时，都可以处理此事件。而不论此事件是否在这一级得到处理，事件都可以停止向上传播或者继续向上传播，这是典型的不纯的责任链模式。

自从 AWT 1.1 版本以后，AWT 的事件处理模型与 1.0 版本相比有了很大的变化。新的事件处理模型是建立在观察者模式的基础之上的，而不再是建立在责任链模式的基础之上的。

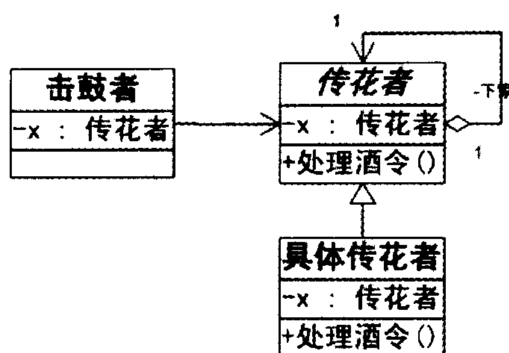
关于新的事件处理模型和观察者设计模式，请见本书“观察者（Observer）模式”一章。

45.5 《红楼梦》中击鼓传花的故事

显然，击鼓传花符合责任链模式的定义。参加游戏的人是一个个具体处理者对象，击鼓的人便是客户端对象。花代表酒令，是传向处理者的请求，每一个参加游戏的人在接到传来的花时，可选择的行为只有两个：一是将花向下传；一是执行酒令——喝酒。一个人不能既执行酒令，又向下家传花。当某一个人执行了酒令之后，游戏重新开始。击鼓的人并不知道最终是由哪一个做游戏的人执行酒令，当然执行酒令的人必然是做游戏的人们中的一个。

当某个成员执行了酒令后，游戏就从其下家重新开始，因此，责任链的起点是动态的。

击鼓传花的类图结构如下图所示。

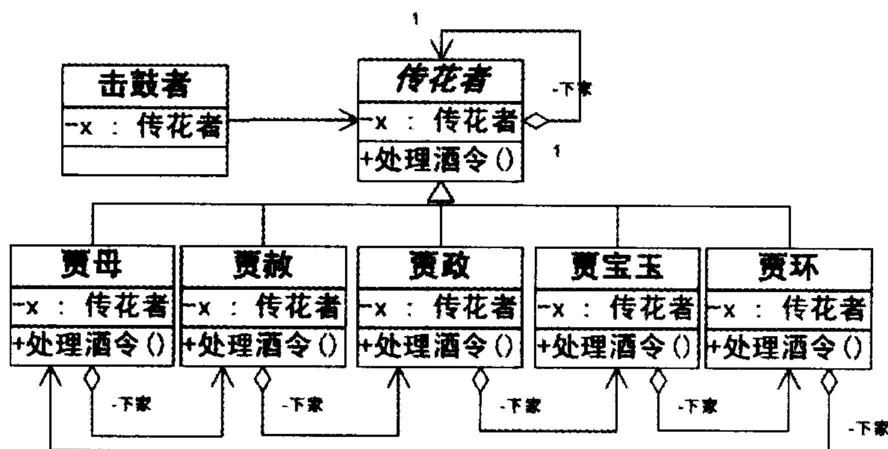


单独考虑击鼓传花系统，那么像贾母、贾赦、贾政、贾宝玉和贾环等传花者均应当是同一个“具体传花者”的对象，而不应当是独立的“具体传花者”类。但是责任链模式往往是建立在现有系统的基础之上的，因此，链的结构和组成不由责任链模式本身决定。

系统的分析

在《红楼梦》第七十五回里生动地描述了贾府里的一场击鼓传花游戏：“贾母坐下、左垂首贾赦、贾珍、贾琏、贾蓉；右垂首贾政、宝玉、贾环、贾兰、团团围坐。……贾母便命折一枝桂花来，命一媳妇在屏后击鼓传花。若花到谁手中，饮酒一杯……于是先从贾母起，次贾赦，一一接过。鼓声两转，恰恰在贾政手中住了，只得饮了酒。”这场游戏接着又把花传到了宝玉和贾赦手里，接着又传到了贾环手里，如此反复。

如果用一个对象系统描述贾府，那么贾母、贾赦、贾政、贾宝玉和贾环等就应当分别由一个个具体类代表，按照责任链模式，这场击鼓传花游戏的类图应当如下图所示。

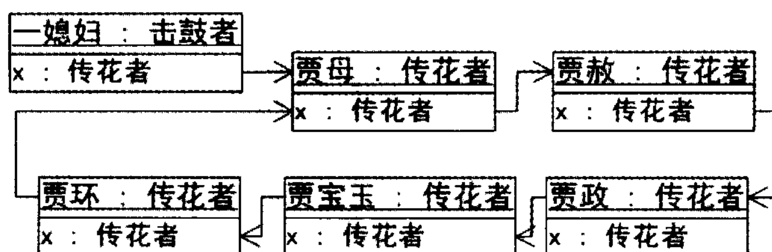


换言之，在击鼓传花游戏里，有下面的几种角色：

- 抽象传花者或 Handler 角色：定义出参加游戏的传花人要遵守的规则，也就是一个处理请求的接口和对下家的引用。
- 具体传花者或 ConcreteHandler 角色：每一个传花者都知道下家是谁，要么执行酒令，要么把花向下传。这个角色由贾母、贾赦、贾珍、贾琏、贾蓉、贾政、宝玉、贾环、贾兰等扮演。

- 击鼓人或 Client 角色：即行酒令的击鼓之人。《红楼梦》没有给出此人的具体姓名，只是说由“一媳妇”扮演。

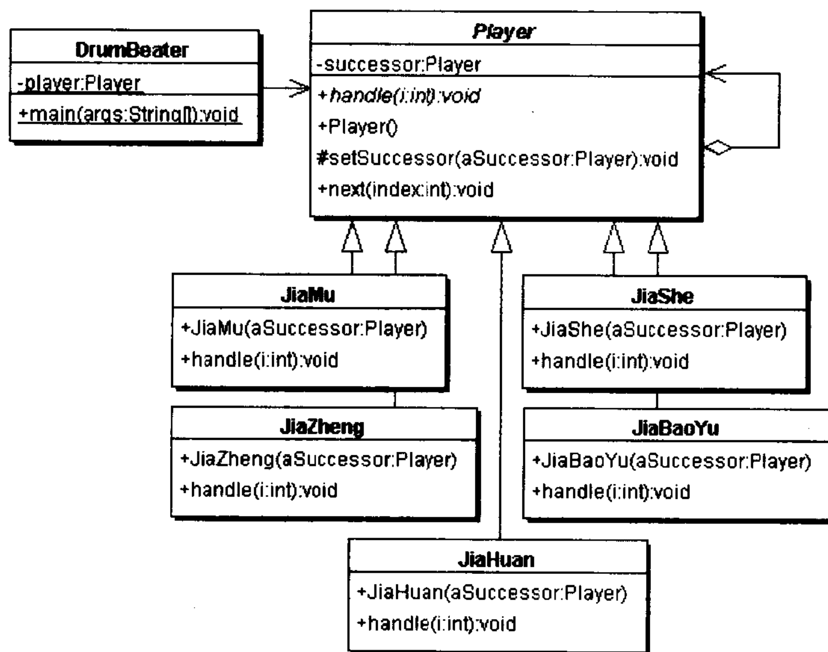
贾府这次击鼓传花的示意性对象图如下图所示。



可以看出，击鼓传花游戏满足责任链模式的定义，是纯的责任链模式的例子。

45.6 Java 系统的解

如下图所示，其中给出了这些类的具体接口设计。读者不难看出，DrumBeater（击鼓者）、Player（传花者）、JiaMu（贾母）、JiaShe（贾赦）、JiaZheng（贾政）、JiaBaoYu（宝玉）、JiaHuan（贾环）等组成了这个系统。



客户端类 DrumBeater 的源代码如代码清单 5 所示。

代码清单 5：DrumBeater 类的源代码

```
public class DrumBeater
{
    private static Player player;
```

```
    static public void main(String[] args)
    {
        // 创建责任链
        player = new JiaMu( new
          new JiaZheng( new JiaBaoYu(new JiaHuan(null)))));
        //规定由第四个处理者处理请求
        player.handle(4);
    }
}
```

下面是抽象类 Player 的源代码，如代码清单 6 所示。这个类声明了三个重要的方法：

（1）请求处理方法 handle()，这是一个抽象方法。

（2）setSuccessor()方法，可以设定下家，这是一个具体方法。

（3）向下家传花的方法 next()，这是一个具体方法。

代码清单 6：抽象传花者 Player 类的源代码

```
abstract class Player
{
    /**
     *    请求处理方法，调用此方法处理请求
     */
    abstract public void handle(int i);

    private Player successor;

    /**
     *    默认构造子
     */
    public Player()
    {
        successor = null;
    }

    /**
     *    赋值方法，调用此方法设定下家
     */
    protected void setSuccessor(Player aSuccessor)
    {
        successor = aSuccessor;
    }

    /**
     *    公开方法，将“花”传给下家；如果没有下家，系统停止运行
     */
    public void next(int index)
    {
```

```
        // 判断下家对象是否有效
    if( successor != null )
        {
            // 将请求传给下家
        successor.handle(index);
        }
        else
        {
            //系统停止运行
        System.out.println("Program terminated.");
            System.exit(0);
        }
    }
}
```

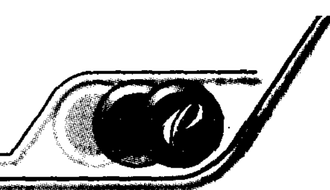
抽象类 Player 给出了两个方法的实现：一个是 setSuccessor()，另一个是 next()。前者用来设置一个传花者对象的下家，后者用来将酒令传给下家。Player 类给出了一个抽象方法 handle()，代表执行酒令。

下面介绍的是这些具体传花者类将给出 handle()方法的实现。首先是代表贾母的 JiaMu 类的源代码，如代码清单 7 所示。此类继承自抽象传花者角色，给出了请求处理方法 handle() 的具体实现。

代码清单 7：代表贾母的 JiaMu 类的源代码

```
class JiaMu extends Player
{
    /**
     *    构造子，以下家对象为参量
     */
    public JiaMu(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
     *    请求处理方法，调用此方法处理请求
     */
    public void handle(int i)
    {
        if( i == 1 )
        {
            System.out.println("Jia Mu gotta drink!");
        }
        else
        {
            System.out.println("Jia Mu passed!");
            next(i);
```



```
        }
    }
}
```

下面是具体类 JiaShe 的源代码，如代码清单 8 所示，此类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现。

代码清单 8：代表贾赦的 JiaShe 类的源代码

```
class JiaShe extends Player
{

    /**
     *     构造子，以下家对象为参量
     */
    public JiaShe(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
     *     请求处理方法，调用此方法处理请求
     */
    public void handle(int i)
    {
        if( i == 2 )
        {
            System.out.println("Jia She gotta drink!");
        }
        else
        {
            System.out.println("Jia She passed!");
            next(i);
        }
    }
}
```

下面是具体类 JiaZheng 的源代码，如代码清单 9 所示。此类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现。

代码清单 9：代表贾政的 JiaZheng 类的源代码

```
class JiaZheng extends Player
{
    /**
     *     构造子，以下家对象为参量
     */
    public JiaZheng(Player aSuccessor)
    {
```

```
        this.setSuccessor(aSuccessor);
    }

    /**
     *    请求处理方法，调用此方法处理请求
     */
    public void handle(int i)
    {
        if( i == 3 )
        {
            System.out.println("Jia Zheng gotta drink!");
        }
        else
        {
            System.out.println("Jia Zheng passed!");
            next(i);
        }
    }
}
```

下面是具体类 JiaBaoYu 的源代码，如代码清单 10 所示。此类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现。

代码清单 10：代表贾宝玉的 JiaBaoYu 类的源代码

```
class JiaBaoYu extends Player
{
    /**
     *    构造子，以下家对象为参量
     */
    public JiaBaoYu(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
     *    请求处理方法，调用此方法处理请求
     */
    public void handle(int i)
    {
        if( i == 4 )
        {
            System.out.println("Jia Bao Yu gotta drink!");
        }
        else
        {
            System.out.println("Jia Bao Yu passed!");
            next(i);
```



```
        }
    }
}
```

下面是具体类 JiaHuan 的源代码，如代码清单 11 所示。此类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现。

代码清单 11：代表贾环的 JiaHuan 类的源代码

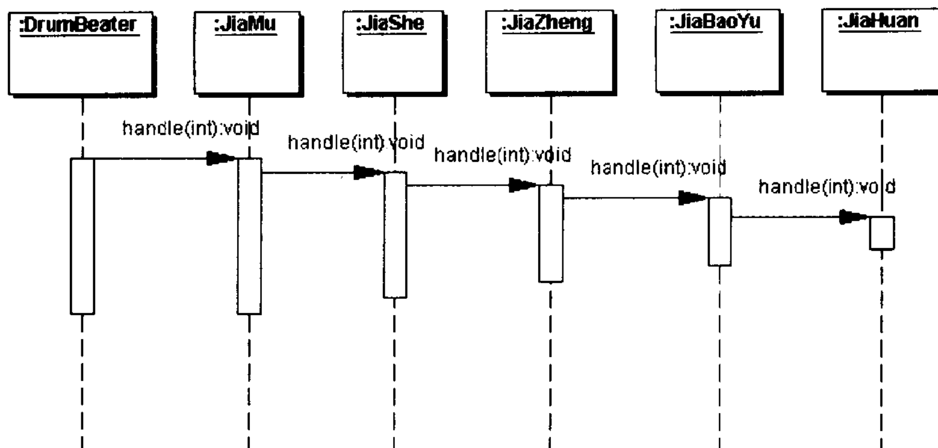
```
class JiaHuan extends Player
{
    /**
     *     构造子，以下家对象为参量
     */
    public JiaHuan(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
     *     请求处理方法，调用此方法处理请求
     */
    public void handle(int i)
    {
        if( i == 5 )
        {
            System.out.println("Jia Huan gotta drink!");
        }
        else
        {
            System.out.println("Jia Huan passed!");
            next(i);
        }
    }
}
```

可以看出，DrumBeater 设定了责任链的成员和他们的顺序：责任链由贾母开始到贾环，周而复始。JiaMu 类、JiaShe 类、JiaZheng 类、JiaBaoYu 类与 JiaHuan 类均是抽象传花者 Player 类的子类。

本节所实现的 DrumBeater 类在把请求传给贾母时，实际上指定了由 4 号传花者处理酒令。虽然 DrumBeater 并不知道哪一个传花者类持有号码 4，但是这个号码在本系统一开始就是写死的。这当然并不符合击鼓传花游戏的精神，因为这个游戏实际上要求有两个同时进行的过程：击鼓过程和传花过程。击鼓应当是定时停止的，当击鼓停止时，执行酒令者就确定了。但是本节这样做可以使问题得到简化并将读者的精力放在责任链模式上，而不是两个过程的处理上。

击鼓传花的活动时序图如下图所示。



下一章会给出一个多线程的系统，更加逼真地模拟击鼓传花系统。

在下面的情况下可以使用责任链模式：

（1）系统已经有一个由处理者对象组成的链。这个链可能由合成模式给出。

（2）有多于一个的处理者对象会处理一个请求，而且事先并不知道到底由哪一个处理者对象处理一个请求。这个处理者对象是动态确定的。

（3）系统想发出一个请求给多个处理者对象中的某一个，但是不明显指定是哪一个处理者对象会处理此请求。

（4）处理一个请求的处理者对象集合需要动态地指定时。

责任链模式降低了发出命令的对象和处理命令的对象之间的耦合，它允许多于一个的处理者对象根据自己的逻辑来决定哪一个处理者最终处理这个命令。换言之，发出命令的对象只是把命令传给链结构的起始者，而不需要知道到底是链上的哪一个节点处理了这个命令。

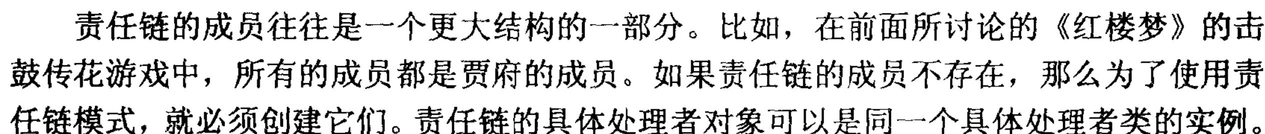
显然，这意味着在处理命令时，允许系统有更多的灵活性。哪一个对象最终处理一个命令可以因为由哪些对象参加责任链，以及这些对象在责任链上的位置不同而有所不同。

45.7 责任链模式的实现

链结构的由来

值得指出的是，责任链模式并不创建出责任链。责任链的创建必须由系统的其他部分创建出来。

责任链模式减低了请求的发送端和接收端之间的耦合，使多个对象都有机会处理这个请求。一个链可以是一条线，一个树，也可以是一个环。链的拓扑结构可以是单连通的或多连通的，责任链模式并不指定责任链的拓扑结构。但是责任链模式要求在同一个时间里，命令只可以被传给一个下家（或被处理掉），而不可以传给多于一个下家。如下图所示，责任链是一个树结构的一部分。



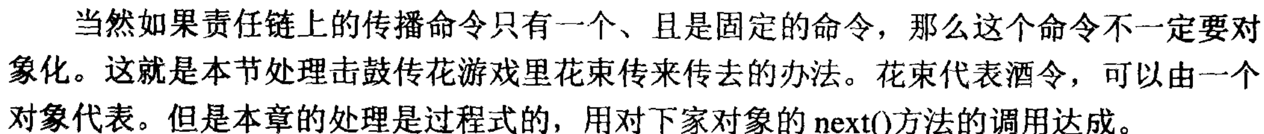
责任链的成员往往是一个更大结构的一部分。比如，在前面所讨论的《红楼梦》的击鼓传花游戏中，所有的成员都是贾府的成员。如果责任链的成员不存在，那么为了使用责任链模式，就必须创建它们。责任链的具体处理者对象可以是同一个具体处理者类的实例。

在 Java 1.0 版的 AWT 事件处理模型里，责任链便是视窗上的构件的容器等级结构。

在下面会谈到在 Internet Explorer 的 DHTML 的 DOM 事件处理模型里，责任链则是 DOM 等级结构本身。

命令的传递

在一个责任链上传递的可能不只有一个命令，而是数个命令。这些命令可以采取抽象化层、具体化层的多态性实现方式，如下图所示，从而可以将命令对象与责任链上的对象之间的责任分隔开，并将命令对象与传播命令的对象分隔开。



当然如果责任链上的传播命令只有一个、且是固定的命令，那么这个命令不一定要对象化。这就是本节处理击鼓传花游戏里花束传来传去的办法。花束代表酒令，可以由一个对象代表。但是本章的处理是过程式的，用对下家对象的 next()方法的调用达成。

45.8　对象的树结构

读者可以在本书“合成（Composite）模式”一章中找到关于对象树结构的一般性的讨论。

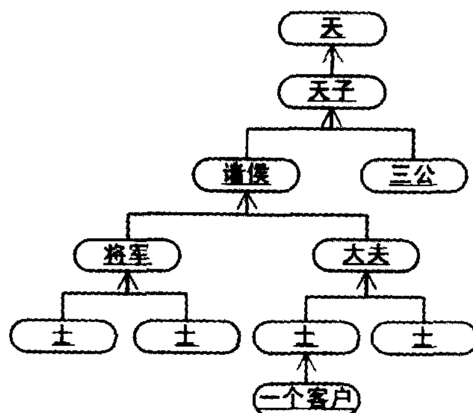
在面向对象的技术里，对象的树结构是一个强有力的工具，更是模式理论的一个重要的组成部分，可以应用到复合模式、装饰模式和迭代子模式中去。

《墨子.天志》说：“庶人竭力从事，未得次己而为政，有士政之，士竭力从事，未得次己而为政，有将军、大夫政之；将军、大夫竭力从事，未得次己而为政，有三公、诸侯

政之；三公、诸侯竭力听治，未得次己而为政，有天子政之；天子未得次己而为政，有天政之。”

“次”意为恣意。上面的话就是说，百姓有官吏管治，官吏由将军和士大夫管治，将军和士大夫由三公和诸侯管治，三公和诸侯由天子管治，天子由天管治。

墨子论责任和责任链的传播如下图所示。图中有阴影的对象给出了一个可能的责任链选择。



当一个百姓提出要求时，此要求会传达到“士”一级，再到“大夫”一级，进而传到“诸侯”一级，“天子”一级，最后到“天”一级。

45.9 DHTML 中的事件处理

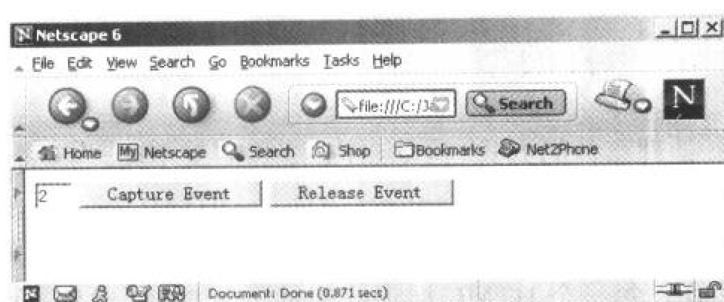
浏览器的 DOM（Document Object Model）模型中的事件处理均采用责任链模式。本节首先考察 Netscape 浏览器的 DHTML 的事件处理，然后再研究 Internet Explorer 的事件模型。

Netscape 的事件模型

Netscape 的事件处理机制叫做“事件捕捉”（Event Capturing）。在事件捕捉机制里，一个事件是从 DOM 的最高一层向下传播，也就是说，Window 对象是第一个接到事件上的，然后是 document 对象，如此往下。因此事件的产生对象反而是最后一个接到事件上的。

如果要使一个对象捕获某一个事件，只需要调用 captureEvent()方法；如果要使一个对象把某一个事件向下传而不处理此事件，只需要对此对象使用 releaseEvents 方法即可。下面考察一个简单的事件捕获和传递的例子，如下图所示。





在这个例子里，有一个 textbox 和两个 button，一个叫做“Capture Event”，单击后会使网页的 click 事件被捕捉，文字框中的计数会加一；另一个叫做“Release Event”，单击后会使网页的 click 事件不被捕捉。

使 click 事件被捕捉需要调用 captureEvent()方法，而使 click 事件不被捕捉需要调用 releaseEvent()方法。下面是具体的 HTML 和 JavaScript 代码，如代码清单 12 所示。

代码清单 12：JavaScript 和 HTML 源代码

```
<html>
<head>
<script language="JavaScript1.2">
  var counter = 0;
  function captureClick()
  {
    document.captureEvents(Event.CLICK);
    document.onclick = clickHandler;
  }
  function clickHandler()
  {
    document.myForm.myText.value = counter++;
  }
  function releaseClick()
  {
    document.releaseEvents(Event.CLICK);
    document.onclick = "";
  }
  </script>
  </head>
  <body>
  <form name="myForm">
  <input type=TEXT size=2 value="" value="" name="myText">
  <input type=BUTTON value="Capture Event" onmousedown='captureClick()'>
  <input type=BUTTON value="Release Event" onmousedown='releaseClick()'>
  </form>
  </body>
  </html>
```

显然，一个事件可以在几个不同的等级上得到处理，这是一个不纯的责任链模式。

Internet Explorer 的事件模型

Internet Explorer 处理事件的方式与 Netscape 既相似又不同。当一个事件发生在 Internet Explorer 所浏览的网页中时，Internet Explorer 会使用 DHTML 的 "Event Bubbling"，即事件浮升机制处理此事件。Internet Explorer 的 DOM 模型是 HTML 对象等级结构和事件处理机制。在 DOM 里面，每一个 HTML 标识符都是一个 DOM 对象，而每一个 DOM 对象都可以产生事先定义好的几个事件中的一个（或几个）。这样的一个事件会首先发生在事件所属的对象上，然后向上传播，传到此对象所属的容器对象上。因此，事件浮升机制恰恰是事件捕捉机制的相反面。

在 Event Bubbling 机制里，产生事件的对象首先会收到事件。然后，事件会依照对象的等级结构向上传播。比如，一个 DIV 里有一个 Form，Form 里面又有一个 Button，那么当 Button 的 onclick 事件产生时，Form 的 onclick 事件代码就会被执行。然后，事件就会传到 DIV 对象。如果 DIV 对象的 onclick 事件有任何代码的话，这个代码就会被执行，然后事件继续沿着 DOM 结构上行。

如果要阻止事件继续向上传播，可以在事件链的任何一个节点上把 cancelBubble 性质设置成 True 即可。

Internet Explorer 浏览器几乎为所有的 HTML 标识符都提供了事件句柄，因此 Internet Explorer 不需要 captureEvents()方法和 releaseEvents()方法来捕获和释放事件。下面的 JavaScript 语句指定了 document 对象的 onclick 事件的处理方法：

```
document.onclick = functionName;
```

而下面的语句则停止了 document 对象对 onclick 事件的处理：

```
document.onclick = null;
```

因为事件处理性质被赋值 null，document 便没有任何的方法处理此事件。换言之，null 值禁止了此对象的事件处理。这种方法可以用到任何的对象和任何的事件上面。当然这一做法不适用于 Netscape。

与在 Netscape 中一样，一个事件处理方法可以返还 Boolean 值。比如，单击一个超链接标识符是否造成浏览器跟进，取决于此超链接标识符的 onclick 事件是否返还 true。

为了显示 Internet Explorer 中的事件浮升机制，本节特准备了下面的例子。一个 Form 里面有一个 Button，如下图所示。



其 HTML 代码如代码清单 13 所示。

代码清单 13：JavaScript **和** HTML **源代码**

```
<html>
<body onclick="alert('document receives onclick event');">
  <form name="myForm" onclick="alert('myForm receives onclick event');"
    <input type=button name=myButton value="Click Here"
      onclick="alert('myButton receives onclick event');">
  </form>
</body>
</html>
```

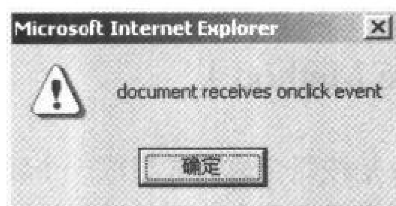
当 myButton 的 onclick 事件发生时，myButton 的事件处理首先被激发，从而显示出如下图所示的对话框。



然后事件会像气泡一样浮升到上一级的对象，即 myForm 对象上。myForm 对象的事件处理给出如下图所示的对话框。



这以后的事件继续浮升到更上一级的对象即 body 上。这时，document 对象的事件处理被激发，并给出如下图所示的对话框。



这就是事件浮升（Event Bubbling）机制。

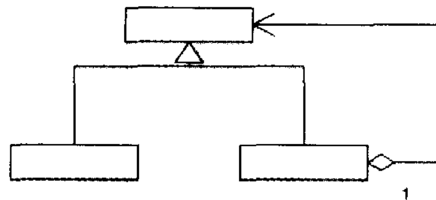
显然，这三级对象组成一个责任链，而事件便是命令或请求。当事件沿着责任链传播时，责任链上的对象可以选择处理或不处理此事件。不论事件在某一个等级上是否得到处理，事件都可以停止上浮或继续上浮，这是不纯的责任链模式。

45.10　责任链模式与其他模式的关系

责任链模式与以下的设计模式相关。

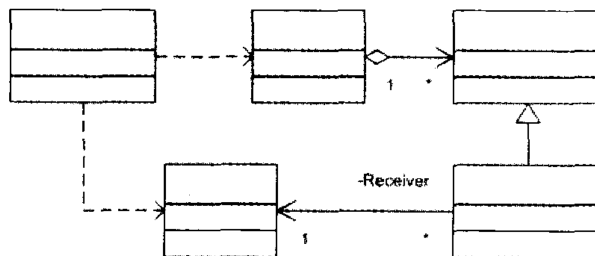
合成模式（Composite Pattern）

当责任链模式中的对象链属于一个较大的结构时，这个较大的结构可能是按照合成模式的要求构造出来的。合成模式的简略类图如下图所示。

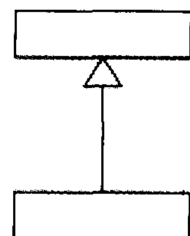


命令模式（Command Pattern）

责任链模式使一个特定的请求接收对象对请求或命令的执行变得不确定。而命令模式使得一个特定的对象对一个命令的执行变得明显和确定。命令模式的简略类图如下图所示。



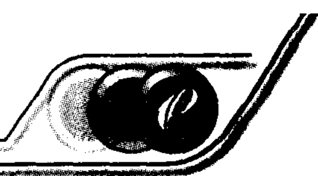
模版方法模式（Template Method）



当组成责任链的处理者对象是按照合成模式组成一个较大结构的责成部分时，模版方法模式经常用来组织单个的对象的行为。模版方法模式的简略类图如右图所示。

问答题

1. 在称为“拱猪”的纸牌游戏中，四个参加者中有“猪”牌的，可以选择一个时机放出这张“猪”牌。“猪”牌放出后，四个人中的一个会不可避免地拿到这张“猪”牌。



请使用责任链模式说明这一游戏，并给出 UML 结构图。

2. 《墨子·迎敌祠》里描述守城军队的结构："城上步一甲、一戟，其赞三人。五步有伍长，十步有什长，百步有佰长，旁有大帅，中有大将，皆有司吏卒长。"

一个兵勇需要上级批准以便执行一项任务，他要向伍长请求批准。伍长如果有足够的权限，便会批准或驳回请求；如果他没有足够的权限，便会向上级即什长转达这个请求。什长便会重复同样的过程，直到大将那里。一个请求最终会被批准或驳回，然后就会向下传，最后传回到发出请求的兵勇手里。

有些请求会很快返回，有些则要经过较长的过程。请求到底由谁批准，事前并不知道。请求的处理者并不是固定的，有些军官会晋升、转业或从别的单位转过来等。

请使用责任链模式解释这个核准请求的系统。

（本例受文献[ALPERT98]里"Chain of Responsibility"一节所给出的一个例子的启发。）

3. 王羲之在《兰亭序》中写道："有清流激湍，映带左右，引以为流觞曲水，列坐其次。"讲的是大家列坐水畔，随水流放下带羽毛的酒杯饮酒。远道而来的酒杯流到谁的面前，谁就取而饮之。

在这个活动中，参加者坐成一排，面对着一条弯曲的小溪。侍者把酒杯盛满酒，让酒杯沿着小溪向下漂流。酒杯漂到一个参加者面前的时候，他可以选择取酒饮之，也可以选择让酒杯漂向下家。

假设每一杯酒最终都会被参加者中之一喝掉，那么这个游戏是不是纯的责任链模式呢？

4. 请考虑四人接力赛跑。当裁判吹响哨子时，第一棒赛跑者率先跑出，然后将接力棒传给第二棒赛跑者；第二棒赛跑者继续跑，然后将棒传给第三棒赛跑者；最后是第四棒的赛跑者。

请问这符合什么设计模式？请给出一个四人接力比赛的模拟系统设计类图和对象图。

问答题答案

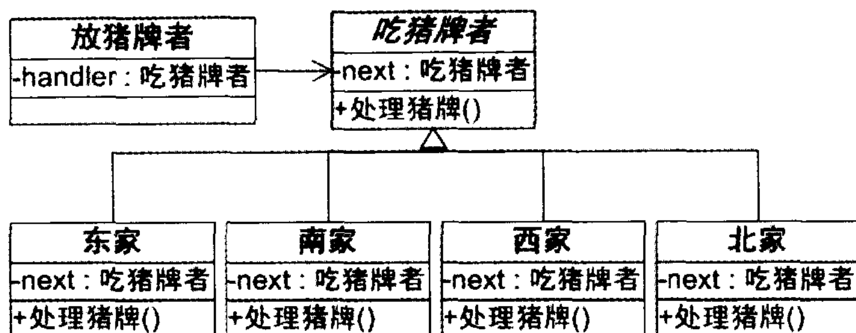
1. 这是一个纯的责任链模式。

首先，在"猪"牌放出之后，每个人都要么躲过"猪"牌，要么吃住"猪"牌。"猪"牌便是责任链模式中的请求，四个人便是四个处理者对象，组成责任链。

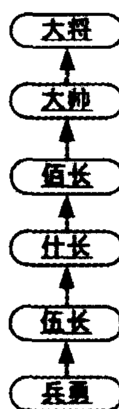
每一个参加者的行为不仅仅取决于他手中的牌，而且取决于他是否想得"猪"牌。一个想收全红的人，可能会全力揽"猪"牌，一个不想收全红的人，一般不想收"猪"牌，除非他想阻止别人收"猪"牌。因为一旦有人收全红，另外三个人就会付出较大的代价，因此阻止别人收全红的动机，会促使一个参与者主动收"猪"牌。有的时候，放出"猪"牌的人也会想要得"猪"牌而得不到，有的时候放出"猪"牌的人想要害人但却害了自己。

这就是说，到底是四个人中的哪一个人得到"猪"牌，是完全动态决定的。

系统的 UML 结构图如下图所示。



由于玩牌的时候，可能有四人位置的任意调换，或者有候补者在旁等待，一旦在任的玩家被淘汰，便可上任。这样四个人组成的牌局是动态变化的。同时因为谁会拿到“猪”牌在每一局均会不同，因此谁会放出“猪”牌也是动态的。



因此，责任链的组成和顺序并不是一成不变的，而是动态的和变化的。

2. 墨子的守城部队的等级结构可以用如右图所示的对象图表示。

显然，这是一个纯的责任链模式。

兵勇在发出申请时不知道他的申请会向上传播多少等级。

任何提出申请的兵勇便是客户端，伍长、什长、佰长、大帅和大将是责任链的具体处理者对象。

一个申请会在链上传播，直到某一级有合适的权限的军官处理申请为止。每一个申请必会得到处理：批准或驳回。一个被处理过的申请会按照相反的方向传播，最后传回到发出申请的兵勇手中。

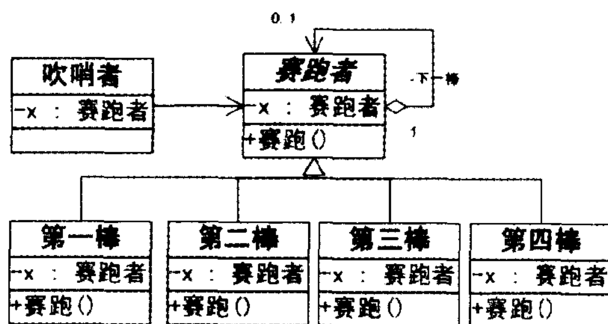
3. 这是纯的责任链模式。

首先，酒便是请求的代表。每一个酒会的参与者都是一个请求的处理者对象，所有的参加者组成责任链。一个酒杯会漂过每一个参加者，代表一个请求经过每一个请求处理者对象。

每一个酒会的参加者都有可能选择喝掉某一杯酒，或者让酒杯继续漂向下一个参加者。假定所有的酒最后都会被某一个参加者喝掉，因此这是纯的责任链模式。

4. 这不是纯的责任链模式。

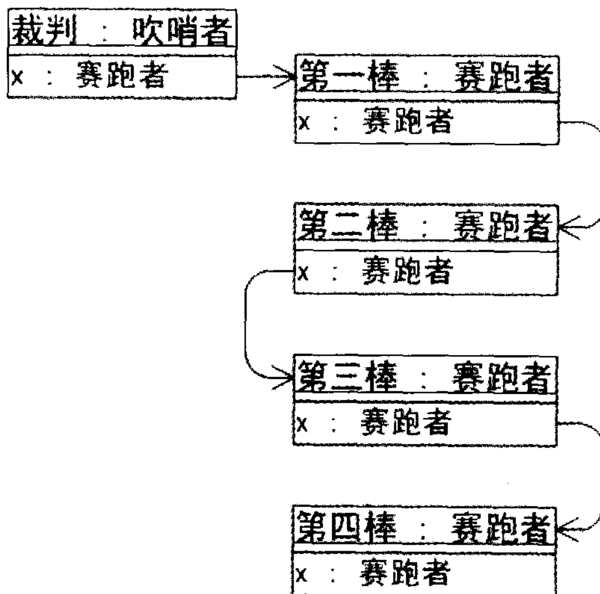
首先，接力棒便是代表请求，处理请求就是跑步。四个赛跑者全都是请求的处理者对象，他们组成责任链。接力棒从一个赛跑者传给下一个赛跑者，代表请求被每一个请求处理者对象传给下一个请求处理者对象。这个体育项目的模拟系统设计图如下图所示。





每一个类中都有一个私有变量 x。“吹哨者”的变量 x 指向“第一棒”，而“第一棒”的变量 x 指向“第二棒”，“第二棒”的变量 x 指向“第三棒”，“第三棒”的变量 x 指向“第四棒”，“第四棒”的变量 x 的值为 null。在图中的抽象赛跑者的聚合关系是从 1 到 0 而不是从 1 到 1，就是因为“第四棒”的原因。

接力棒在比赛期间总是被某一个赛跑者所持有，但是每一个赛跑者都会处理请求。而且除了最后一个赛跑者之外，每一个赛跑者又都将请求传给下一个人，因此，这不是纯的责任链模式，这个系统的对象图如下图所示。



参考文献

[WYKE99]R.A. Wyke, J.D. Gilliam, and Charlton Ting. Pure JavaScript. A Code-Intensive Premium Reference, SAMS, 1999

第 46 章　专题：定时器与击鼓传花

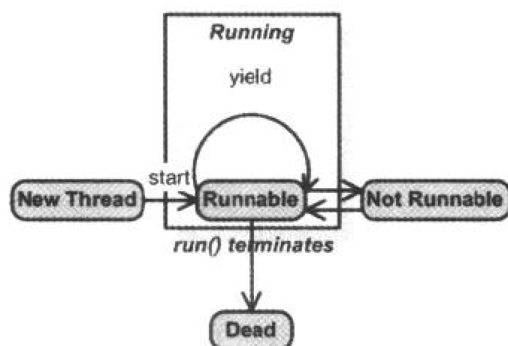
"责任链（Chain of Responsibility）模式"一章谈到在《红楼梦》第七十五回里描述的一场击鼓传花游戏。仔细想一想就会发现，击鼓的活动和传花的活动应当是两个独立的线程：击鼓线程开始运行，代表游戏开始；传花的线程开始运行，代表传花的人们开始传花；击鼓结束表示时间到了，传花的活动便立即停止，代表传花的线程则需要执行一项任务。

46.1　线　　程

线程是可以在一个程序里面平行运行的执行路线，线程与线程之间可以共享变量。Java 线程的启动很快，而且程序员可以比控制过程更容易地控制线程。使用线程可以使程序快速地处理同时来自不同计算机的请求，因此在分散系统（Distributed Systems）设计里，常常要使用线程。

本章简单地复习线程的生命周期作为"热身"。

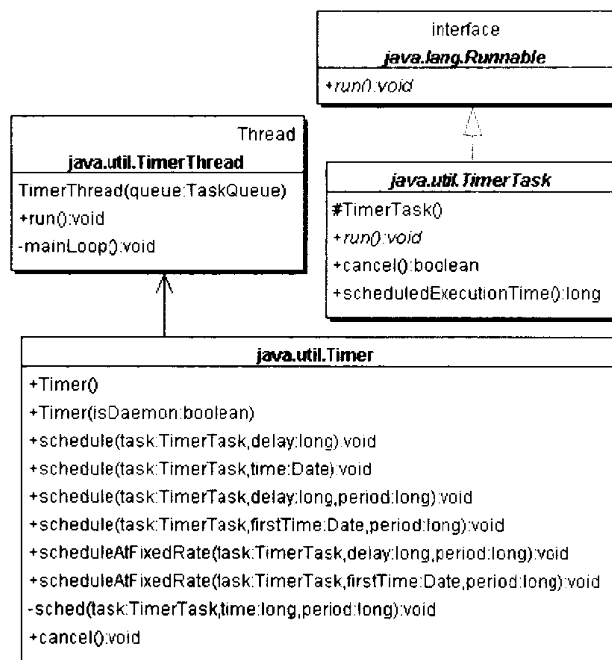
首先，当一个线程被创建时，线程处于"New Thread"状态，这时线程还没有运行。然后，客户端需要调用 start()方法，使线程开始运行。调用这个方法会为此线程产生一个新的运行环境，并调用 run()方法，线程开始运行后便进入 Runnable 状态。其次，通过调用 sleep()、wait()或者由于 I/O 阻塞，可以使线程进入 Not Runnable 状态。最后，如果线程处于 Runnable 或 Not Runnable 状态的话，调用 isAlive()方法会得到 true。如果 run()方法结束，线程可以自己停下来，线程也可以由外界停下来。用 UML 状态图描述线程的生命周期，如下图所示。



在一个多线程的程序里面，为避免线程无谓地等待（starvation），甚至线程之间的死锁（deadlock），多线程的编程特别要注意线程的同步化（synchronization）。

46.2 定 时 器

线程的编程比较不易掌握，所以除非特别需要，应当尽量使用高级线程 API。比如 java.util.Timer 类便是 Java 1.3 版本引入的定时器类，此类封装了定时器所需的线程功能。与此类相应的，还有一个 TimerTask 类，封装了当被定时的任务需要反复发生时所需的线程功能。如果仅仅是为了定时执行一个操作，应当使用 Timer 类。用 UML 类图描述 Timer 和 TimerTask 类的设计，如下图所示。



考察一下 Timer 和 TimerTask 的类图便可看出，Timer 是 TimerThread 的子类，而 TimerThread 是 Thread 的实现，因此 Timer 是 Thread 的实现。同时，TimerTask 是 Runnable 的实现，从而也是一个线程。这就是说，Timer 和 TimerTask 都是线程，而且是彼此独立的线程。

Timer 类给出一系列的 scheduler 方法，用于定时执行一个 TimerTask 类型的对象。TimerTask 的对象则含有具体的 run()方法，这个方法可以调用系统所要执行的内部或外部任务。

为了演示这个高级 API 使用方法，本章给出下面的例子，使用 java.util.Timer 在 5 秒钟后执行一个任务，这个任务封装在 RemindTask 对象里面，如代码清单 1 所示。

代码清单 1：使用 Timer 的例子

```
import java.util.Timer;
import java.util.TimerTask;
public class Reminder
{
```



```
    Timer timer;
    /**
     *     构造子，在 seconds 秒后执行一个任务
     */
    public Reminder(int seconds)
    {
        timer = new Timer();
        timer.schedule(new RemindTask(),
            seconds * 1000);
    }
    /**
     *     内部成员类，描述将要执行的任务
     */
    class RemindTask extends TimerTask
    {
        public void run()
        {
          System.out.println("Time's up!");
          //Terminate the timer thread
          timer.cancel();
        }
    }
    /**
     *     静态方法
     */
    public static void main(String args[])
    {
        System.out.println("About to schedule task.");
        new Reminder(5);
        System.out.println("Task scheduled.");
    }
}
```

在每一个 Timer 对象的背后，都有一个背景线程，用来顺序执行 Timer 的任务。这些任务应当迅速结束，不然就会霸占 timer 用来执行任务的线程，从而延迟后面的任务。

当所有的任务都完成后，timer 的任务执行线程就会顺利地结束并被 Java 垃圾收集器所收集，只是垃圾收集的时间并不能预测。如果程序想要迅速地结束 timer 的任务执行线程，调用方就应当调用 timer 的 cancel()方法。上面的例子就使用了这一办法。

Timer 类是线程安全的，换言之，可以有多个线程共享同一个 Timer 对象，而不需要同步化。这是使用 Timer 的一个好处。



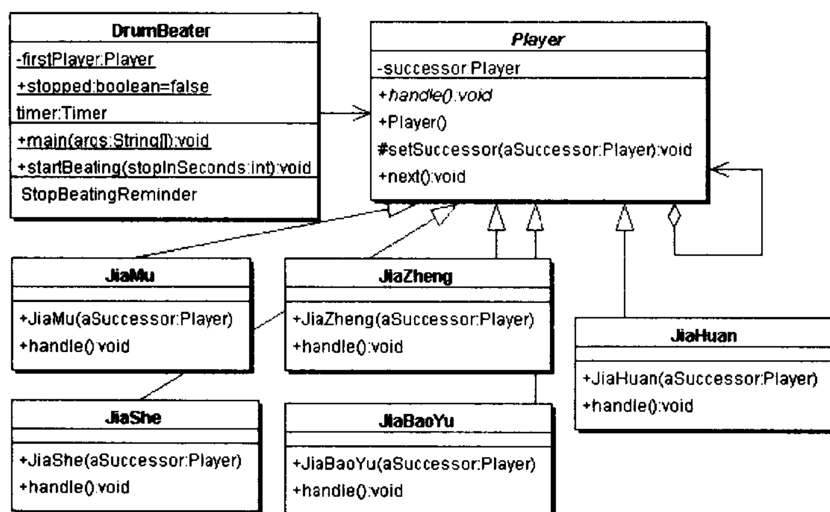
Timer 类实际上是用 Object.wait(long)方法实现定时的，因此，Timer 类不能保证实时效果。Timer 类可以处理多达几千个任务。

46.3 击鼓传花

前面给出了击鼓传花游戏的模拟实现，同时也指出哪个实现是不够逼真的。本章准备给出一个更为逼真的模拟实现。

击鼓传花的游戏其实应当使用两个线程来描述，一个线程用于模拟击鼓，另一个用于模拟传花。击鼓的过程只持续一段时间，因此可以使用 Timer 来定时执行。换言之，可以定时执行一个"停止击鼓"的操作。击鼓停止对于所有参加传花的宾客而言，是一个公共的信号，它意味着传花立即停止。当传花停止时，手中拿着花的宾客就应执行酒令。然后游戏重新开始。

如下图所示，便是本章改进后的模拟击鼓传花游戏的设计图。



可以看出，这个设计图与前面一章给出的系统设计图十分相似，只是各个角色的实现有细节上的改变。抽象类 Player 的改变是 next()方法的实现，这个改进的实现消除了事先决定执行酒令者的设计缺陷。

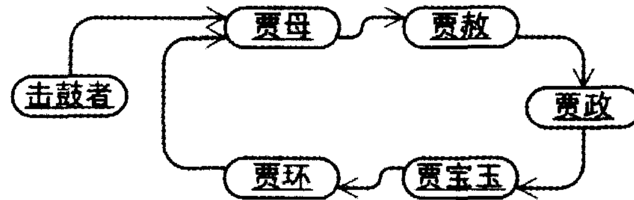
在这个改进后的设计里面，DrumBeater 类含有一个内部类，作为 TimerTask 的实现。DrumBeater 的源代码如代码清单 2 所示。

代码清单 2：DrumBeater 的源代码

```
import java.lang.Thread;
import java.util.Timer;
import java.util.TimerTask;
public class DrumBeater
{
    private static Player firstPlayer;
    public static boolean stopped = false;
    Timer timer;
    /**
```

```
     *    静态方法
     */
    static public void main(String[] args)
    {
        DrumBeater drumBeater =
            new DrumBeater();
        JiaMu jiaMu = new JiaMu(null);
        jiaMu.setSuccessor( new JiaShe (
            new JiaZheng(
            new JiaBaoYu(
            new JiaHuan( jiaMu ) ) ) ) );
        // 开始击鼓过程
        drumBeater.startBeating(1);
        // 由贾母开始传花
        firstPlayer = jiaMu;
        firstPlayer.handle();
    }
    /**
     *    调用下面的方法开始击鼓过程
     */
    public void startBeating(int stopInSeconds)
    {
        System.out.println("Drum beating started...");
        timer = new Timer();
        timer.schedule(new StopBeatingReminder(),
            stopInSeconds * 1000);
    }
    /**
     *    内部成员类，描述停止击鼓的任务
     */
    class StopBeatingReminder extends TimerTask
    {
        public void run()
        {
            System.out.println("Drum beating stopped!");
            stopped = true;
            //Terminate the timer thread
            timer.cancel();
        }
    }
}
```

可以看出，DrumBeater 对象指定了一个责任链：由贾母到贾赦，由贾赦到贾政，由贾政到贾宝玉，由贾宝玉到贾环，再由贾环回到贾母，形成一个环形，模拟对象图如下图所示，其源代码如代码清单 3 所示。



由抽象传花者的源代码可以看出，这是一个 Java 抽象类，声明了三个重要的方法：

- 请求处理方法 handle()，这是一个抽象方法。
- setSuccessor()方法，可以设定下家，这是一个具体方法。
- 向下家传花的方法 next()，这是一个具体方法。

代码清单 3：对象 Player 的源代码

```
abstract class Player
{
    abstract public void handle();

    private Player successor;

    /**
     *    默认构造子
     */
    public Player()
    {
        successor = null;
    }

    /**
     *    赋值方法，调用此方法设定下家
     */
    protected void setSuccessor(Player aSuccessor)
    {
        successor = aSuccessor;
    }

    /**
     *    公开方法，将“花”传给下家
     *    如果没有下家，系统停止运行
     */
    public void next()
    {
        // 判断下家对象是否有效
        if( successor != null )
        {
            // 将请求传给下家
            successor.handle();
```

```
        }
        else
        {
            //系统停止运行
            System.out.println("Program is terminating.");
            System.exit(0);
        }
    }
}
```

代表贾母的 JiaMu 类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现，如代码清单 4 所示。

代码清单 4：贾母由 JiaMu 类代表

```
class JiaMu extends Player
{
    /**
     *    构造子，以下家对象为参量
     */
    public JiaMu(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
     *    请求处理方法，调用此方法处理请求
     */
    public void handle()
    {
        // 检查击鼓是否已经停止
        if( DrumBeater.stopped )
        {
            // 执行酒令
            System.out.println("Jia Mu gotta drink!");
        }
        else
        {
            System.out.println("Jia Mu passed!");
            // 将花传给下家
            next();
        }
    }
}
```

代表贾赦的 JiaShe 类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现，如代码清单 5 所示。

代码清单 5：贾赦由 JiaShe 类代表

```
class JiaShe extends Player
{
    /**
     *    构造子，以下家对象为参量
     */
    public JiaShe(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
     *    请求处理方法，调用此方法处理请求
     */
    public void handle()
    {
        // 检查击鼓是否已经停止
        if( DrumBeater.stopped )
        {
            // 执行酒令
            System.out.println("Jia She gotta drink!");
        }
        else
        {
            System.out.println("Jia She passed!");
            // 将花传给下家
            next();
        }
    }
}
```

代表贾政的 JiaZheng 类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现，如代码清单 6 所示。

代码清单 6：贾政由 JiaZheng 类代表

```
class JiaZheng extends Player
{
    /**
     *    构造子，以下家对象为参量
     */
    public JiaZheng(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
```

```
     *     请求处理方法，调用此方法处理请求
     */
    public void handle()
    {
        // 检查击鼓是否已经停止
        if( DrumBeater.stopped )
        {
            // 执行酒令
            System.out.println("Jia Zheng gotta drink!");
        }
        else
        {
            System.out.println("Jia Zheng passed!");
            // 将花传给下家
            next();
        }
    }
}
```

代表贾宝玉的 JiaBaoYu 类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现，如代码清单 7 所示。

代码清单 7：贾宝玉由 JiaBaoYu 类代表

```
class JiaBaoYu extends Player
{
    /**
     *     构造子，以下家对象为参量
     */
    public JiaBaoYu(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
     *     请求处理方法，调用此方法处理请求
     */
    public void handle()
    {
        // 检查击鼓是否已经停止
        if( DrumBeater.stopped )
        {
            // 执行酒令
            System.out.println("Jia Bao Yu gotta drink!");
        }
        else
        {
            System.out.println("Jia Bao Yu passed!");
```

```
            // 将花传给下家
            next();
        }
    }
}
```

代表贾环的 JiaHuan 类继承自抽象传花者角色，给出了请求处理方法 handle()的具体实现，如代码清单 8 所示。

代码清单 8：贾环由 JiaHuan 类代表

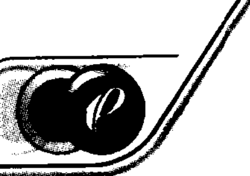
```
class JiaHuan extends Player
{
    /**
     *    构造子，以下家对象为参量
     */
    public JiaHuan(Player aSuccessor)
    {
        this.setSuccessor(aSuccessor);
    }

    /**
     *    请求处理方法，调用此方法处理请求
     */
    public void handle()
    {
        // 检查击鼓是否已经停止
        if( DrumBeater.stopped )
        {
            // 执行酒令
            System.out.println("Jia Huan gotta drink!");
        }
        else
        {
            System.out.println("Jia Huan passed!");
            // 将花传给下家
            next();
        }
    }
}
```

程序运行的情况如代码清单 9 所示。

代码清单 9：运行的结果

```
Drum beating started...
Jia Mu passed!
Jia She passed!
Jia Zheng passed!
```



```
Jia Bao Yu passed!
......
Jia Huan passed!
Drum beating stopped!
Jia Mu gotta drink!
```

可以看出，当击鼓开始后，参加传花者开始传递花束，传递的方式是调用下家的 next()方法。

传花的人们在接到花束时，首先会检查击鼓者的状态，也就是 stopped 性质。如果 stopped 是 true，那么传花人就执行酒令；不然就调用下家的 next()方法，象征着将花束传到下家。

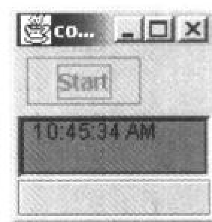
问答题

1. 当一个类已经有自己的父类，不能继承 Thread 类时，怎样实现一个线程类？
2. 继续上题，请利用线程的概念写一个浏览器 Applet，实现数字钟的功能。
3. 每一个线程都是一个线程组的成员。在默认情况下，线程处于哪一个线程组里面？
4. 请考察 java.lang.ThreadGroup 类有什么操作？
5. 请问 JVM 的 scheduler 是否是固定优先权的 scheduler？
6. 每一个线程都有一个优先权，请问优先权是越大越好还是越小越好？
7. 当两个线程的优先权相同时，scheduler 会怎么选择？
8. 请设计一个简单的 scheduler，在一个确定的时刻执行一个 DOS 命令。

问答题答案

1. 当一个类已经有自己的父类，不能继承 Thread 类时，可以实现 Runnable 接口，提供一个 run()方法来做成一个线程。

2. 数字钟 Applet 的运行情况如右图所示。



Applet 的源代码如代码清单 10 所示。

代码清单 10：数字钟 Applet 的源代码

```
import java.awt.Graphics;
import java.awt.Color;
import java.util.*;
import java.text.DateFormat;
public class Clock extends java.applet.Applet
    implements Runnable
{
    private Thread clockThread = null;
    //Applet 初始化方法
    public void init()
    {
        setBackground(Color.green);
    }
```

```
    //Applet 启动方法
    public void start()
    {
        if (clockThread == null)
        {
            clockThread = new Thread(this, "Clock");
            //thread starts running
            clockThread.start();
        }
    }
    //一个 Thread 方法
    public void run()
    {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread)
        {
            repaint(); //causes paint() to be called
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e){ }
        }
    }
    //一个 applet 方法
    public void paint(Graphics g)
    {
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        DateFormat dateFormatter = DateFormat.getTimeInstance();
        g.drawString(dateFormatter.format(date), 5, 10);
    }
    // 将 Applet 的停止方法置换掉
    public void stop()
    {
        clockThread = null;
    }
}
```

3. 在默认情况下，线程与创建它的线程属于同一个线程组。

4. java.lang.ThreadGroup 有如下的操作：

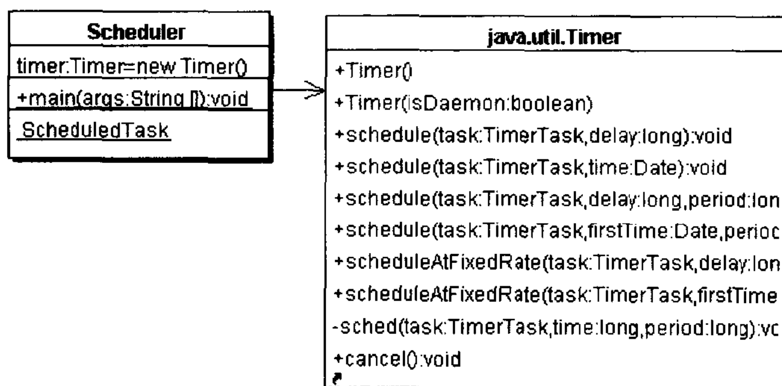
   1. getMaxPriority and setMaxPriority
   2. getDaemon and setDaemon
   3. getName
   4. getParent and parentOf
   5. toString

5. JVM 的 scheduler 是固定优先权的 scheduler。

6. 每一个线程都有一个优先权，优先权越大越好。scheduler 或选择具有最高优先权的 runnable 线程执行。当此线程停止、yield 或变成 Not Runnable 时，一个低优先权的线程开始执行。

7. 当两个线程的优先权相同时，scheduler 轮流选择两个线程执行。

8. Scheduler 系统的设计类图如下图所示。



在系统中使用 java.util.Timer 对象，以及一个类型为 java.util.TimerTask 的子类对象，其源代码如代码清单 11 所示。

代码清单 11：Scheduler 类的源代码

```
import java.util.*;
import java.io.IOException;
public class Scheduler
{
    private static Timer timer = new Timer();
     /**
      *     静态方法
      */
    public static void main(String []args)
    {
        Calendar calendar = Calendar.getInstance();
        calendar.set(Calendar.HOUR_OF_DAY, 10);
        calendar.set(Calendar.MINUTE, 30);
        calendar.set(Calendar.SECOND, 30);
        Date time = calendar.getTime();
        timer.schedule(new ScheduledTask(), time);
        System.out.println("A job has been scheduled at "
              + time);
    }
     /**
      *     静态内部成员类
      */
    static class ScheduledTask extends TimerTask
```

```
    {
        public void run()
        {
          try
          {
              String command = "notepad c:/boot.ini";
              Process child = Runtime.getRuntime().exec(command);
              System.out.println("The scheduled job has been executed.");
          }
          catch (IOException e)
          {
              System.out.println("Exception " + e);
          }
        }
    }
}
```

参考文献

[JAVATUT00]The Java Tutorial：A Short Course on the Basics. 3rd edition (January 15，2000) by Mary Campione，Kathy Walrath，Alison Huml. Addison-Wesley

第 47 章　命令（Command）模式

命令（Command）模式属于对象的行为模式[GOF95]。命令模式又称为行动（Action）模式或交易（Transaction）模式。

命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

47.1 引　　言

命令模式是对命令的封装。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。

每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

命令允许请求的一方和接收请求的一方能够独立演化，从而具有以下的优点：

（1）命令模式使新的命令很容易地被加入到系统里。

（2）允许接收请求的一方决定是否要否决（Veto）请求。

（3）能较容易地设计一个命令队列。

（4）可以容易地实现对请求的 Undo 和 Redo。

（5）在需要的情况下，可以较容易地将命令记入日志。

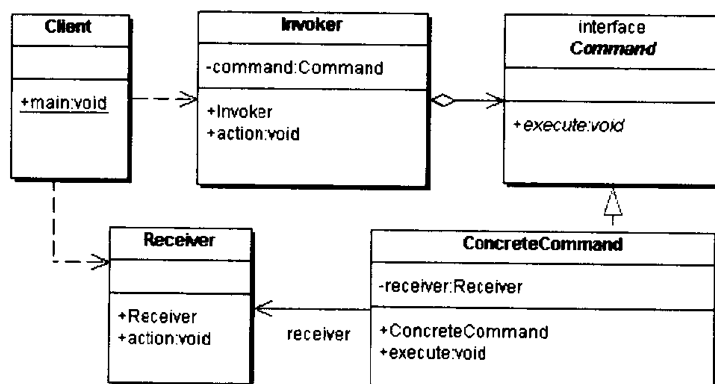
对熟悉 C 语言的读者，命令模式便是面向对象形式的回呼（Callback）。Java 语言虽然不支持函数指针，但是命令模式可以达到同样的目的。

命令模式不是新的发明，在美猴王大闹天宫之前就有了。那时玉帝命令太白金星召美猴王上天："金星径入（水帘洞）当中，面南立定道：'我是西方太白金星，奉玉帝招安圣旨，下界请你上天，拜受仙录。'"玉帝是系统的客户端，太白金星是命令的发出者，猴王是命令的接收者，圣旨就是命令。玉帝的这一道命令就是要求猴王到上界报到。玉帝只管发出命令，而不管命令是怎样传达到美猴王的。太白金星负责将圣旨传到，可是美猴王怎么执行圣旨、何时执行圣旨是美猴王自己的事。果不然，不久美猴王就大闹了天宫。

47.2 命令模式的结构

命令模式的类图

下面以一个示意性的系统为例，说明命令模式的结构。这个系统的结构图如下图所示。



命令模式涉及到五个角色，它们分别是：

- 客户（Client）角色：创建了一个具体命令（ConcreteCommand）对象并确定其接收者。
- 命令（Command）角色：声明了一个给所有具体命令类的抽象接口。这是一个抽象角色，通常由一个 Java 接口或 Java 抽象类实现。
- 具体命令（ConcreteCommand）角色：定义一个接收者和行为之间的弱耦合；实现 execute()方法，负责调用接收者的相应操作。execute()方法通常叫做执行方法。
- 请求者（Invoker）角色：负责调用命令对象执行请求，相关的方法叫做行动方法。
- 接收者（Receiver）角色：负责具体实施和执行一个请求。任何一个类都可以成为接收者，实施和执行请求的方法叫做行动方法。

下面就是客户角色的源代码，如代码清单 1 所示。客户角色创建了 Receiver 对象、ConcreteCommand 对象和 Invoker 对象。

代码清单 1：客户角色的源代码

```
public class Client
{
    public static void main(String[] args)
    {
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        Invoker invoker = new Invoker( command );
        invoker.action();
```

```
    }
}
```

下面给出 Invoker 类的源代码，如代码清单 2 所示。Invoker 类的构造子接收 Command 类型的参量，并提供一个 action()方法作为行动方法。这个 action()方法调用 Command 对象的 execute()方法执行此命令对象。

代码清单 2：请求者（Invoker）角色的源代码

```
public class Invoker
{
    private Command command;
   /**
    *  构造子
    */
    public Invoker(Command command)
    {
        this.command = command;
    }
   /**
    *  行动方法
    */
    public void action()
    {
        command.execute();
    }
}
```

Receiver 类是命令的接收者，在命令的控制下执行行动方法。这里仅给出一个 action()方法的示意性实现，如代码清单 3 所示。读者可以在把命令模式应用到自己的系统设计中时，按照需要给出多个行动方法和具体细节。

代码清单 3：接收者（Receiver）角色的源代码

```
public class Receiver
{
   /**
    *  构造子
    */
    public Receiver()
    {
        //write code here
    }
   /**
    *  行动方法
    */
    public void action()
    {
```

```
            System.out.println("Action has been taken.");
        }
}
```

抽象命令角色规定出具体命令对象必须实现的接口，即一个执行方法，如代码清单 4 所示。

代码清单 4：抽象命令角色由 Command 接口扮演

```
public interface Command
{
    /**
     *  执行方法
     */
    void execute();
}
```

具体命令类实现了抽象命令角色所规定出的接口，即一个执行方法，如代码清单 5 所示。

代码清单 5：具体命令类的源代码

```
public class ConcreteCommand implements Command
{
    private Receiver receiver;
    /**
     *  构造子
     */
    public ConcreteCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }
    /**
     *  执行方法
     */
    public void execute()
    {
        receiver.action();
    }
}
```

命令模式的活动序列

命令模式的活动序列如下所示：

（1）客户端创建了一个 ConcreteCommand 对象，并指明了接收者。

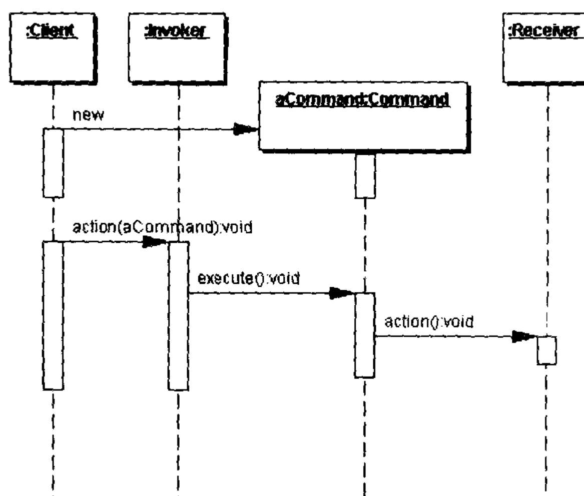
（2）请求者对象保存了 ConcreteCommand 对象。

（3）请求者对象通过调用 action()方法发出请求。如果命令是能撤销（undo）的，那

么 ConcreteCommand 保存了调用 execute()方法之前的状态。

（4）ConcreteCommand 对象调用接收的一方的方法执行请求。

命令模式的活动时序图如下图所示。



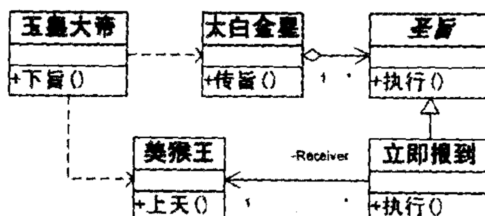
关于命令的撤销（undo）和恢复（redo），请见下一节的讨论。

47.3　玉帝传美猴王上天

玉皇大帝传美猴王上天的过程，使用命令模式来理解是非常恰当的，这里面涉及到五个角色：

- 命令（Command）角色：即抽象圣旨，给出所有的具体圣旨都必须具备的接口。这是一个抽象角色，在这里由一个 Java 接口实现。
- 具体命令（ConcreteCommand）角色：即具体圣旨，是玉皇大帝宣美猴王上天报到的那份圣旨。这个具体圣旨给出玉皇大帝和美猴王作为客户端和命令的接收者之间的弱耦合。execute()方法便是圣旨的执行方法。
- 请求者（Invoker）角色：请求者太白金星调用此角色的行动方法，要求美猴王上天。
- 接收者（Receiver）角色：便是美猴王。美猴王的上天报到的操作必须由美猴王自己做出。普天之下的万物都可以成为玉帝圣旨的接收者。
- 客户（Client）角色：由玉皇大帝扮演。玉帝创建了一个具体命令对象（就是圣旨），玉帝还要指明圣旨的接收者是美猴王，并把命令交给太白金星传达。

这个模拟系统的设计如下图所示。



玉皇大帝采用命令模式来处理圣旨显然是比较聪明的做法。如果不采用命令模式，玉帝便要自己处理所有的操作。在这里便包括美猴王怎样上天和怎样报到的一切细节。

换言之，使用了命令模式，玉皇大帝便不需要直接和美猴王打交道，而把打交道的细节交给太白金星去处理。圣旨封装了玉皇大帝的命令，以及命令所代表的操作。

47.4 Java 语言内的例子：AWT 的事件处理

Java 语言使用了命令模式处理 java.awt 库的事件委派处理模型。在这个事件处理模型里面，命令对象实现 AWT 的 listener 接口，相当于命令接口。为把一个命令对象与一个 AWT 构件连接起来，需要把它登记成一个事件的 listener。构件只认识 listener 接口，而不在乎接口是怎么实现的。换言之，AWT 构件不在意命令所代表的实际操作。

如果一个视窗系统的命令都由按键构件代表，那么为了彰显命令模式，每一个按键都给出一个实现 ActionListener 接口的类，那么这些按键类便是具体命令类，而 ActionListener 接口便是抽象命令角色。

但是，AWT 系统并不提供命令模式的撤销和恢复机制。为了支持撤销和恢复，就要在每一个按键构件类里存储局部的状态信息，记住命令的效果。由于 AWT 系统不提供这样的机制，因此需要程序自己实现这一机制。

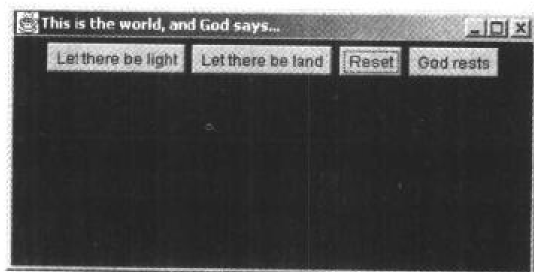
关于命令的撤销和恢复，请见下一节。

47.5 一个例子：创世纪系统

本节使用 java.awt 库的事件处理模型来模拟创世纪的（一部分）过程。这个系统叫做 TheWorld 系统。

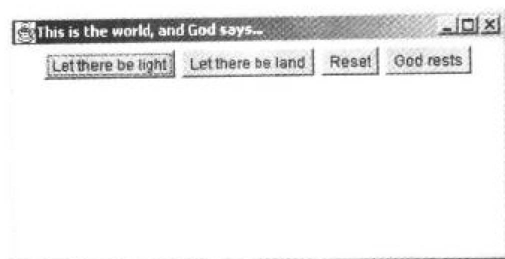
系统是什么样的

在系统开始时，世界是一个黑暗的视窗，如下图所示。

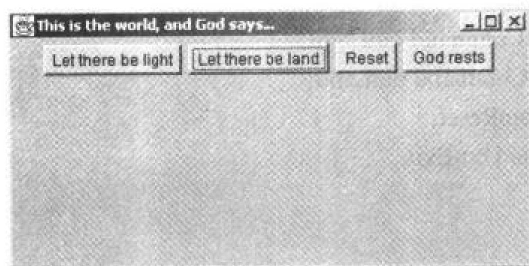


视窗带有四个按键：

- Let There Be Light（要有光！）：这个按键按下之后，世界窗口一片光明，如下图所示。



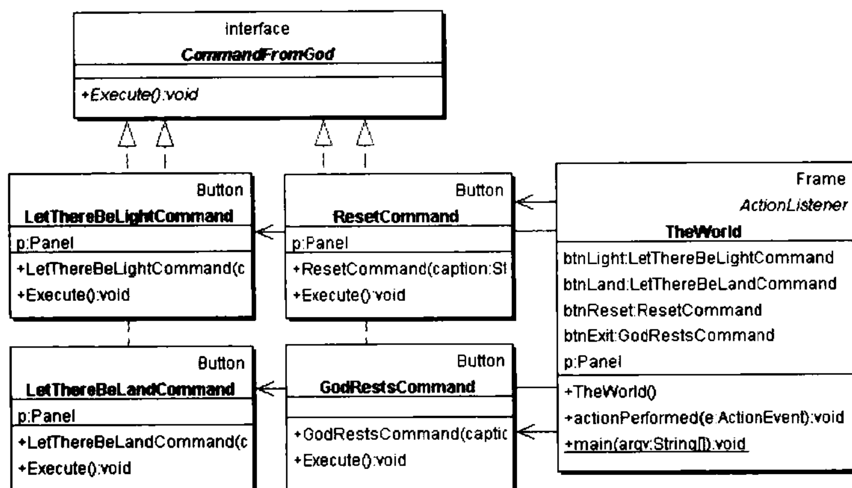
● Let There Be Land（要有地！）：这个按键按下之后，世界充满黄土，如下图所示。



● Reset（复原）：按下这个键时，世界窗口恢复到创世界之前的一片黑暗。
● God Rests（上帝休息了）：按下这个键时，创世界的一天就结束了——系统关闭。

系统的静态结构图

这个世界系统的设计图如下图所示。



显然，在第一天，上帝发出的命令有四个，由视窗的四个按键代表。这四个命令自然要由四个命令对象封装起来。这个世界系统共有六个类/接口。首先是抽象命令角色，由命令接口扮演，如代码清单 6 所示。

代码清单 6：抽象命令角色由命令接口扮演

```
public interface CommandFromGod
{
    public void Execute();
```

```
}
```

其次是请求者角色，即 TheWorld 类，如代码清单 7 所示。上帝是客户端，不在系统里面。

代码清单 7：请求者角色的源代码

```
import java.awt.*;
import java.awt.event.*;
public class TheWorld extends Frame implements ActionListener
{
    private LetThereBeLightCommand btnLight;
    private LetThereBeLandCommand btnLand;
    private ResetCommand btnReset;
    private GodRestsCommand btnExit;
    private Panel p;
    /**
     *  构造子
     */
    public TheWorld()
    {
        super("This is the world, and God says...");
        p = new Panel();
        p.setBackground(Color.black);

        add(p);
        btnLight = new LetThereBeLightCommand("Let there be light", p);
        btnLand = new LetThereBeLandCommand("Let there be land", p);
        btnReset = new ResetCommand("Reset", p);
        btnExit = new GodRestsCommand("God rests");
        p.add(btnLight);
        p.add(btnLand);
        p.add(btnReset);
        p.add(btnExit);
        btnLight.addActionListener(this);
        btnLand.addActionListener(this);
        btnReset.addActionListener(this);
        btnExit.addActionListener(this);
        setBounds(100,100,400,200);
        setVisible(true);
    }
    /**
     *  事件方法
     */
    public void actionPerformed(ActionEvent e)
    {
        Command obj = (Command)e.getSource();
        obj.Execute();
```

```
    }
    static public void main(String[] argv)
    {
        new TheWorld();
    }
}
```

上帝“要有光”的命令封装在具体命令类 LetThereBeLightCommand 里面，如代码清单 8 所示。

代码清单 8：“要有光”的命令类的源代码

```
import java.awt.*;
import java.awt.event.*;
public class LetThereBeLightCommand
    extends Button implements CommandFromGod
{
    private Panel p;
   /**
    *  构造子
    */
    public LetThereBeLightCommand(String caption, Panel pnl)
    {
        super(caption);
        p = pnl;
    }
   /**
    *  执行方法
    */
    public void Execute()
    {
        p.setBackground(Color.white);
    }
}
```

上帝“要有大地”的命令封装在具体命令类 LetThereBeLandCommand 里面，如代码清单 9 所示。

代码清单 9：“要有大地”的具体命令类的源代码

```
import java.awt.*;
import java.awt.event.*;
public class LetThereBeLandCommand
    extends Button implements CommandFromGod
{
    private Panel p;
   /**
    *  构造子
    */
```

```
    public LetThereBeLandCommand(String caption, Panel pnl)
    {
        super(caption);
        p = pnl;
    }
   /**
    *  执行方法
    */
    public void Execute()
    {
        p.setBackground(Color.orange);
    }
}
```

复原命令封装在具体命令类 ResetCommand 里面，如代码清单 10 所示。

代码清单 10：复原命令类的源代码

```
import java.awt.*;
import java.awt.event.*;
public class ResetCommand
    extends Button implements CommandFromGod
{
    private Panel p;
   /**
    *  构造子
    */
    public ResetCommand(String caption, Panel pnl)
    {
        super(caption);
        p = pnl;
    }
   /**
    *  执行方法
    */
    public void Execute()
    {
        p.setBackground(Color.black);
    }
}
```

上帝在忙碌了一天之后，休息了。休息的命令封装在具体命令类 GodRestsCommand 里面，如代码清单 11 所示。

代码清单 11：上帝休息的具体命令类

```
import java.awt.*;
import java.awt.event.*;
public class GodRestsCommand
```

```
    extends Button implements CommandFromGod
{
    /**
     *   构造子
     */
    public GodRestsCommand(String caption)
    {
        super(caption);
    }
    /**
     *   执行方法
     */
    public void Execute()
    {
        System.exit(0);
    }
}
```

在这个系统里面使用了与通常不同的做法：每一个按键对象都来自一个独立的按键类；每一个按键类都封装一个命令所代表的操作——改变世界的颜色。

47.6　一个例子：AudioPlayer 系统

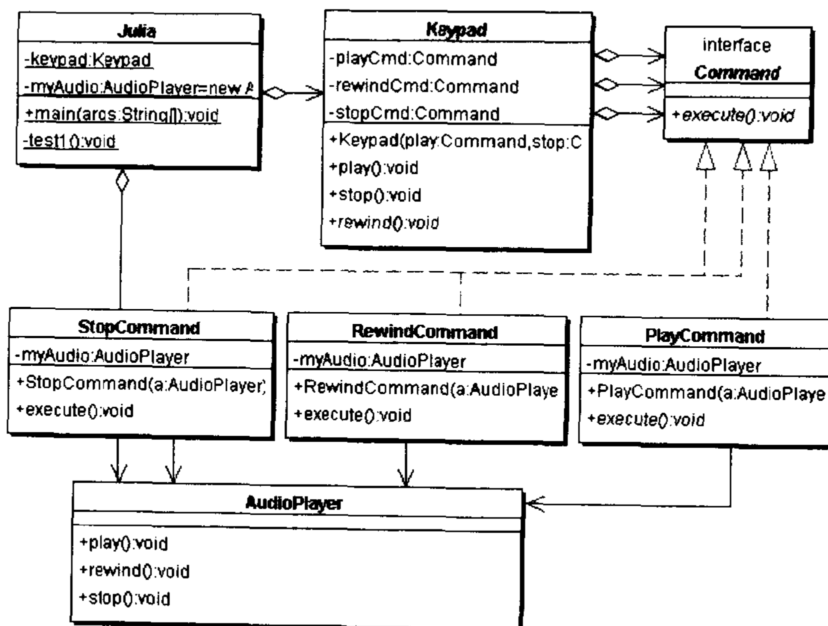
系统的描述

小女孩茱丽（Julia）有一个盒式录音机，本节考虑怎么在 Java 语言中实现这个录音机的模拟系统。此系统有播音（play）、倒带（rewind）和停止（stop）功能，录音机的键盘便是请求者（Invoker）角色；茱丽（Julia）是客户类，而录音机便是接收者角色。Command 类扮演抽象命令角色，而 PlayCommand，StopCommand 和 RewindCommand 便是具体命令类。茱丽（Julia）不需要知道播音（play）、倒带（rewind）和停止（stop）功能是怎么具体执行的，这些命令执行的细节全都由键盘（Keypad）具体实施。茱丽（Julia）只需要在键盘上按下相应的键便可以了。

录音机的设计是典型的命令模式。录音机按键把客户端与录音机的操作细节分割开来。

系统的静态结构

系统的设计如下图所示，即录音机系统的设计图。



这个系统的设计涉及到三个角色：

- 客户端（Client）角色：在这里由茱丽（Julia）类扮演。
- 请求者（Invoker）角色：在这里由键盘（Keypad）类扮演。
- 抽象命令（Command）角色：这是一个抽象角色，由 LayoutManager 类扮演。此角色给出所有的具体 Layout 类所需的接口。
- 具体命令（ConcreteCommand）角色：由 PlayCommand、StopCommand、Stop Command 等扮演，它们封装了不同的录音机的操作。

系统的源代码

下面给出这个模拟系统的源代码。首先，Julia 类扮演系统的客户端角色的源代码如代码清单 12 所示。

代码清单 12：客户端角色 Julia 类的源代码

```
/**
 *    This is the Client role
 */
public class Julia
{
    private static Keypad keypad ;
    private static AudioPlayer myAudio = new AudioPlayer();
    public static void main(String[] args)
    {
        test1();
    }
    private static void test1()
    {
```

```
        Command play = new PlayCommand(myAudio);
        Command stop = new StopCommand(myAudio);
        Command rewind = new RewindCommand(myAudio);
        keypad = new Keypad(play, stop, rewind);
        keypad.play();
        keypad.stop();
        keypad.rewind();
        keypad.stop();
        keypad.play();
        keypad.stop();
    }
}
```

Keypad 类扮演的是 Invoker 角色，它的源代码如代码清单 13 所示。

代码清单 13：请求者（Invoker）角色 Keypad 类的源代码

```
/**
 *  这是接收者角色
 */
public class Keypad
{
    private Command playCmd;
    private Command rewindCmd;
    private Command stopCmd;

    /**
     *  构造子
     */
    public Keypad(Command play, Command stop, Command rewind)
    {
        // concrete Command registers itself with the invoker
        playCmd = play;
        stopCmd = stop;
        rewindCmd = rewind;
    }
    /**
     *  行动方法
     */
    public void play()
    {
        playCmd.execute();
    }
    /**
     *  行动方法
     */
    public void stop()
    {
```

```
            stopCmd.execute();
        }
        /**
         *  行动方法
         */
        public void rewind()
        {
            rewindCmd.execute();
        }
    }
```

抽象命令角色由 Command 接口扮演，源代码如代码清单 14 所示。

代码清单 14：抽象命令接口的源代码

```
    /**
     *  这是抽象命令角色
     */
    public interface Command
    {
        /**
         *  执行方法
         */
        public abstract void execute ( );
    }
```

PlayCommand 扮演了具体命令类角色，它实现了抽象命令角色所给出的接口，如代码清单 15 所示。

代码清单 15：具体命令类 PlayCommand 的源代码

```
    /**
     *  这是一个具体命令
     */
    public class PlayCommand implements Command
    {
        private AudioPlayer myAudio;
        /**
         *  构造子
         */
        public PlayCommand ( AudioPlayer a)
        {
            myAudio = a;
        }
        /**
         *  执行方法
         */
        public void execute( )
        {
```

```
        myAudio.play();
    }
}
```

StopCommand 是另一个具体命令类，其源代码如代码清单 16 所示。

代码清单 16：具体命令类 StopCommand 的源代码

```
/**
 *  这是一个具体命令
 */
public class StopCommand implements Command
{
    private AudioPlayer myAudio;

   /**
    *  构造子
    */
    public StopCommand ( AudioPlayer a)
    {
        myAudio = a;
    }
   /**
    *  执行方法
    */
    public void execute( )
    {
        myAudio.stop();
    }
}
```

RewindCommand 也是一个具体命令类，其源代码如代码清单 17 所示。

代码清单 17：具体命令类 RewindCommand 的源代码

```
/**
 *  这是一个具体命令角色
 */
public class RewindCommand implements Command
{
    private AudioPlayer myAudio;
   /**
    *  构造子
    */
    public RewindCommand ( AudioPlayer a)
    {
        myAudio = a;
    }
   /**
```

```
     *  执行方法
     */
    public void execute()
    {
        myAudio.rewind();
    }
}
```

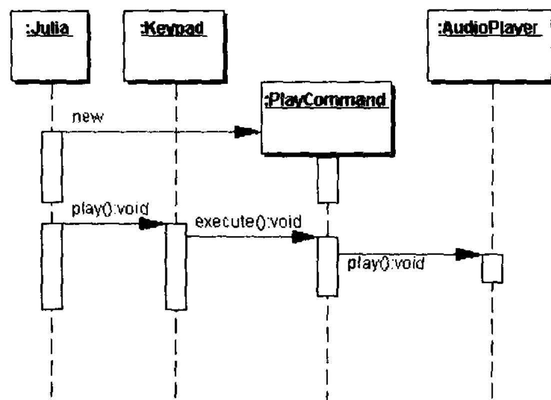
AudioPlayer 类扮演的是 Receiver 角色，其源代码如代码清单 18 所示。

代码清单 18：接收者（Receiver）角色录音机（AudioPlayer）类的源代码

```
/**
 *  此类扮演命令模式的接收者角色
 */
public class AudioPlayer
{
    /**
     *  行动方法
     */
    public void play( )
    {
        System.out.println("Playing...");
    }
    /**
     *  行动方法
     */
    public void rewind( )
    {
        System.out.println("Rewinding...");
    }
    /**
     *  行动方法
     */
    public void stop()
    {
        System.out.println("Stopped.");
    }
}
```

活动序列

录音机系统的活动时序图如下图所示。



系统首先创建茱丽对象，然后创建接收者对象，顺序如下：

（1）茱丽创建一个 PlayCommand 对象；

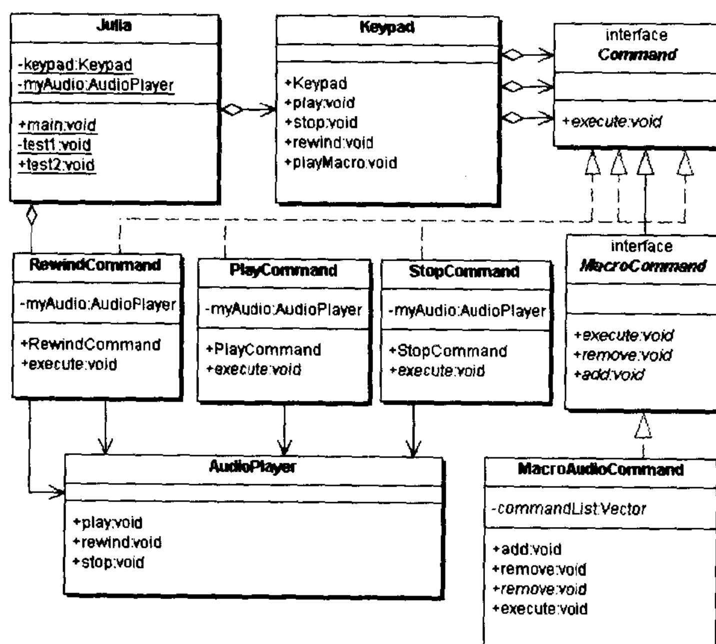
（2）茱丽调用键盘对象的 play()方法；

（3）键盘调用命令对象的 execute()方法；

（4）PlayCommand 对象调用接收端 AudioPlayer 的方法执行请求。

47.7　增加宏命令功能

设想茱丽（Julia）的录音机有一个记录功能，可以把一个一个的命令记录下来，再在任何需要的时候重新把这些记录下来的命令一次性执行，这就是所谓的宏命令集功能。因此，茱丽（Julia）的录音机系统现在有四个键，分别为播音（play）、倒带（rewind）、停止（stop）和宏命令功能。此时系统的设计与前面的设计相比有所增强，主要体现在 Julia 类现在有了一个新的方法，用以操作宏命令键。带有宏命令集功能的录音机模拟系统的设计图如下图所示。



在改造之后，Julia 类需要增加一个新的方法，如代码清单 19 所示。

代码清单 19：客户角色 Julia 类的方法 test2()的源代码

```
......
public static void test2()
{
    Command play = new PlayCommand(myAudio);
    Command stop = new StopCommand(myAudio);
    Command rewind = new RewindCommand(myAudio);
    MacroCommand macro = new MacroAudioCommand();
    macro.add(play);
    macro.add(stop);
    macro.add(rewind);
    macro.add(stop);
    macro.add(play);
    macro.add(stop);
    macro.execute();
}
......
```

系统需要一个代表宏命令的接口，以定义出具体宏命令所需要的接口，如代码清单 20 所示。

代码清单 20：宏命令 MacroCommand 接口的源代码

```
public interface MacroCommand extends Command
{
    /**
     * 执行方法
     */
    void execute();

    /**
     * 宏命令聚集的管理方法
     * 可以删除一个成员命令
     */
    void remove(Command toRemove);

    /**
     * 宏命令聚集的管理方法
     * 可以添加一个成员命令
     */
    void add(Command toAdd);
}
```

具体的宏命令 MacroAudioCommand 类负责把个别的命令合成宏命令，如代码清单 21 所示。

代码清单 21：具体宏命令 MacroAudioCommand 类的源代码

```
import java.util.Vector;
public class MacroAudioCommand implements MacroCommand
{
    private Vector commandList = new Vector();
    /**
     *  宏命令聚集的管理方法
     *  可以添加一个成员命令
     */
    public void add(Command toAdd)
    {
        commandList.addElement(toAdd);
    }
    /**
     *  宏命令聚集的管理方法
     *  可以删除一个成员命令
     */
    public void remove(int index)
    {
        commandList.remove(index);
    }
    public void remove(Command toRemove)
    {
        commandList.removeElement(toRemove);
    }
    /**
     *  执行方法
     */
    public void execute()
    {
        Command nextCommand;
        System.out.println("Automated playback of stored commmands...");
        for (int i=0; i < commandList.size(); i++)
        {
            nextCommand = (Command) commandList.elementAt(i);
            nextCommand.execute();
        }
        System.out.println("Finished automated playback of stored commmands.");
    }
}
```

47.8　模式的实现

模式的实现有几个可以考虑的问题。

首先命令应当“重”一些还是“轻”一些。在不同的情况下，可以做不同的选择。如果把命令设计得“轻”，那么它只是提供了一个请求者和接收者之间的耦合而已，命令代表请求者实现请求。

相反，如果把命令设计的“重”，那么它就应当实现所有的细节，包括请求所代表的操作，而不再需要接收者了。当一个系统没有接收者时，就可以采用这种做法。

更常见的是处于最“轻”和最“重”的两个极端之间的情况。命令类动态地决定调用哪一个接收者类。

其次是否支持 undo 和 redo。如果一个命令类提供一个方法，比如叫 unExecute()，以恢复其操作的效果，那么命令类就可以支持 undo 和 redo。具体命令类需要存储状态信息，包括：

（1）接收者对象实际上实施请求所代表的操作；

（2）对接收者对象所作的操作所需要的参数；

（3）接收者类的最初的状态。接收者必须提供适当的方法，使命令类可以通过调用这个方法，以便接收者类恢复原有状态。

如果只需要提供一层的 undo 和 redo，那么系统只需要存储最后被执行的那个命令对象。如果需要支持多层的 undo 和 redo，那么系统就需要存储曾经被执行过的命令的清单，清单能允许的最大的长度便是系统所支持的 undo 和 redo 的层数。沿着清单逆着执行清单上的命令的反命令（unExecute()）便是 undo；沿着清单顺着执行清单上的命令便是 redo。

47.9 在什么情况下应当使用命令模式

在下面的情况下应当考虑使用命令模式：

- 使用命令模式作为“回呼”在面向对象系统中的替代。“回呼”讲的便是先将一个函数登记上，然后在以后调用此函数。
- 需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。这时命令的接收者可以是在本地，也可以在网络的另外一个地址。命令对象可以在串形化之后传送到另外一台机器上去。
- 系统需要支持命令的撤消（undo）。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用 undo()方法，把命令所产生的效果撤销掉。命令对象还可以提供 redo()方法，以供客户端在需要时，再重新实施命令的效果。
- 如果一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志里读回所有的数据更新命令，重新调用 execute()方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。
- 一个系统需要支持交易（transaction）。一个交易结构封装了一组数据更新命令。使用命令模式来实现交易结构可以使系统增加新的交易类型。

47.10 使用命令模式的优点和缺点

命令模式有很多的优点和缺点。它的优点有：

- 命令模式把请求一个操作的对象与知道怎么执行一个操作的对象分割开。
- 命令类与其他任何别的类一样，可以修改和推广。
- 你可以把命令对象聚合在一起，合成为合成命令。比如上面的例子里所讨论的宏命令便是合成命令的例子。合成命令是合成模式的应用。请参看合成模式的介绍。
- 由于加进新的具体命令类不影响其他的类，因此增加新的具体命令类很容易。

命令模式的缺点如下：

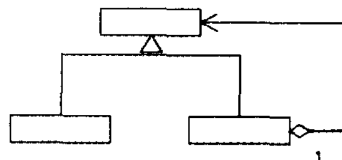
- 使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个，几百个甚至几千个具体命令类，这会使命令模式在这样的系统里变得不实际。

47.11 命令模式与其他模式的关系

和命令模式有关系的其他模式有合成模式、备忘录模式和原始模型模式。

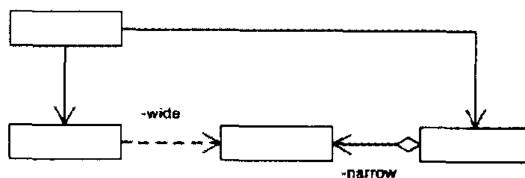
命令模式与合成（Composite）模式的关系

合成模式可以应用到命令类的合成上，从几个具体命令类合成宏命令类。合成模式的简略类图如右图所示。



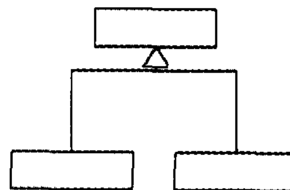
命令模式与备忘录（Memento）模式的关系

如果命令需要撤销（undo）和恢复（redo）功能的话，备忘录模式可以用来存储关于命令的效果状态信息，以便在撤销命令时可以撤销命令的效果。备忘录模式的简略类图如右图所示。



命令模式与原始模型（Prototype）模式的关系

如果命令类带有 clone()方法的话，命令就可以被复制。原始模型模式的简略类图如右图所示。



问答题

1. 假设电视机有三个操作：开机（open）、关机（close）和换台（change channel），可以将其看做是命令。电视机的遥控器相当于请求者（Invoker）的角色，操作者乔（Joe）

相当于此系统的客户，电视机本身就是命令的接收者。请使用 Java 语言设计一个模拟系统，并给出设计图和源代码。

2. 请使用时序图说明上题中对象的活动序列。

3. 股民通过股票经纪人买卖股票就是一个很好的命令模式的例子。请实现一个买卖股票的模拟系统，并说明命令模式怎样优化了系统设计。

4. 请分别给出上面题目中股民买股票和卖股票的时序图。

5. 经理办公室的灯泡坏了，请使用命令模式换一个电灯泡。

6. 如果新灯泡还没有换上去，怎么办？

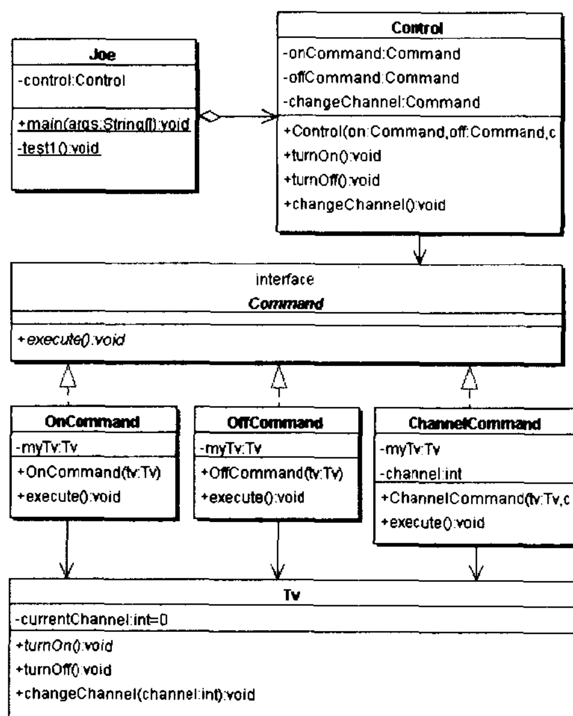
7. 在本节的创世纪系统里面，上帝首先造了光，可谓是电气工程在先，然后是创造黄土大地，土木工程在后。敢问软件工程在上帝的计划里排在什么位置？

8. 诸葛亮在临死前授予杨仪一个锦囊，密嘱曰："我死，魏延必反，……那时自有斩魏延之人也。"同时授马岱以密计，只待魏延喊叫时，便出其不意斩之。杨仪读罢锦囊计策，已知伏下马岱在彼，故依计而行，果然杀了魏延。后人有诗曰："诸葛先机识魏延，已知日后反西川。锦囊遗计人难料，却见成功在马前。"

请问这是不是命令模式，如果是，请给出类图说明；如果不是，请说明原因。

问答题答案

1. 电视机模拟系统的设计如下图所示。



在设计中，Joe 扮演客户角色，电视机的遥控器便是请求者，Joe 通过遥控器发出命令，这些命令由电视机接收。抽象命令角色由一个命令接口扮演，系统一共有三个具体命令类，分别代表三种操作：打开电视、关上电视、换台。显然，遥控器是一个天才的设计，

它把操作电视机的细节从客户端分割开来，Joe 只需要按下相应的按键就可以控制电视机的开、关和换台。

模拟系统的示意性代码如代码清单 22 所示。

代码清单 22：客户角色 Joe 类的源代码

```
public class Joe
{
    private Control control;
    public static void main(String[] args)
    {
        test1();
    }
    private static void test1()
    {
        Tv    myTv = new Tv();
        OnCommand on = new OnCommand(myTv);
        OffCommand off = new OffCommand(myTv);
        ChannelCommand channel = new ChannelCommand(myTv, 2);
        Control control = new Control( on, off, channel);
        control.turnOn();
        control.changeChannel();
        control.turnOff();
    }
}
```

代表遥控器的 Control 类的源代码如代码清单 23 所示。

代码清单 23：客户角色 Julia 类的源代码

```
public class Control
{
    private Command onCommand, offCommand, changeChannel;
   /**
    *  构造子
    */
    public Control(Command on, Command off, Command channel)
    {
        onCommand = on;
        offCommand = off;
        changeChannel = channel;
    }
   /**
    *  行动方法
    */
    public void turnOn()
    {
        onCommand.execute() ;
```

```
    }
    /**
     *  行动方法
     */
    public void turnOff()
    {
        offCommand.execute();
    }
    /**
     *  行动方法
     */
    public void changeChannel()
    {
        changeChannel.execute();
    }
}
```

抽象命令角色的源代码如代码清单 24 所示。

代码清单 24：抽象命令角色 Command 接口的源代码

```
public interface Command
{
    /**
     *  执行方法
     */
    void execute();
}
```

具体命令类 OnCommand 的源代码如代码清单 25 所示。

代码清单 25：具体命令类 OnCommand 的源代码

```
public class OnCommand implements Command
{
    private Tv myTv;
    /**
     *  构造子
     */
    public OnCommand (Tv tv)
    {
        myTv = tv;
    }
    /**
     *  执行方法
     */
    public void execute( )
    {
        myTv.turnOn();
```

```
    }
}
```

具体命令类 OffCommand 的源代码如代码清单 26 所示。

代码清单 26：具体命令类 OffCommand 的源代码

```
public class OffCommand implements Command
{
    private Tv myTv;

   /**
    *  构造子
    */
    public OffCommand (Tv tv)
    {
        myTv   =   tv;
    }
   /**
    *  执行方法
    */
    public void execute()
    {
        myTv.turnOff();
    }
}
```

具体命令类 ChannelCommand 的源代码如代码清单 27 所示。

代码清单 27：ChannelCommand 命令类的源代码

```
public class ChannelCommand implements Command
{
    private Tv myTv;
    private int channel;

   /**
    *  构造子
    */
    public ChannelCommand(Tv tv, int channel)
    {
        myTv = tv;
        this.channel = channel;
    }
   /**
    *  执行方法
    */
    public void execute()
    {
```

```
            myTv.changeChannel(channel);
        }
    }
```

代表电视机的 Tv 类源代码如代码清单 28 所示。

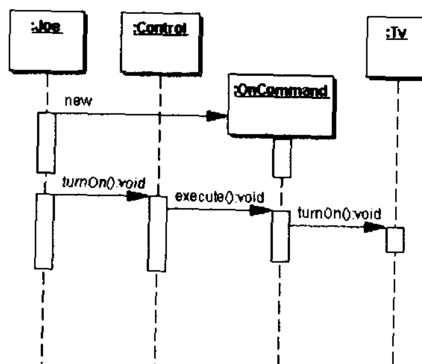
代码清单 28：电视类的源代码

```
public class Tv
{
    private int currentChannel = 0;
    /**
     *  行动方法
     */
    public void turnOn()
    {
        System.out.println("The televisino is on.");
    }

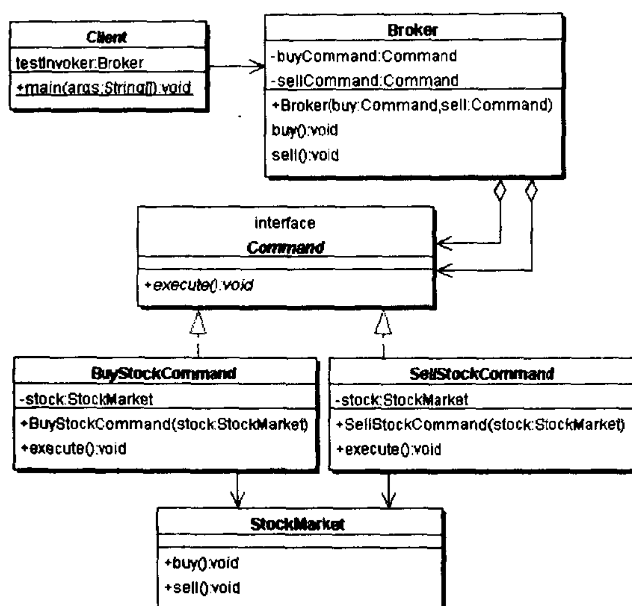
    /**
     *  行动方法
     */
    public void turnOff()
    {
        System.out.println("The television is off.");
    }

    /**
     *  行动方法
     */
    public void changeChannel(int channel)
    {
        currentChannel = channel;
        System.out.println("Now TV channel is " + channel);
    }
}
```

2. 系统的活动时序如下图所示。



3. 股票交易的模拟系统的设计图如下图所示。



在这个系统里，Broker 类代表股票交易员，也就是 Invoker 角色的扮演者，其源代码如代码清单 29 所示；Command 接口便是抽象的命令角色；BuyStockCommand 和 SellStockCommand 是具体命令类；而 StockMarket 类便是命令的接收者。

代码清单 29：Broker 类的源代码

```
public class Broker
{
    private Command buyCommand, sellCommand;

   /**
    *  构造子
    */
    public Broker( Command buy, Command sell)
    {
        buyCommand = buy;
        sellCommand = sell;
    }
   /**
    *  行动方法
    */
    void buy()
    {
        buyCommand.execute() ;
    }
   /**
    *  行动方法
    */
```

```
    void sell()
    {
        sellCommand.execute();
    }
}
```

抽象命令角色的源代码如代码清单 30 所示。

代码清单 30：抽象命令角色 Command 接口的源代码

```
public interface Command
{
    public abstract void execute ( );
}
```

具体命令类 BuyStockCommand 的源代码如代码清单 31 所示。

代码清单 31：具体命令类 BuyStockCommand 的源代码

```
class BuyStockCommand implements Command
{

    private StockMarket stock;

   /**
    *  构造子
    */
    public BuyStockCommand ( StockMarket stock)
    {
        stock = stock;
    }
   /**
    *  执行方法
    */
    public void execute( )
    {
        stock.buy( );
    }
}
```

具体命令类 SellStockCommand 的源代码如代码清单 32 所示。

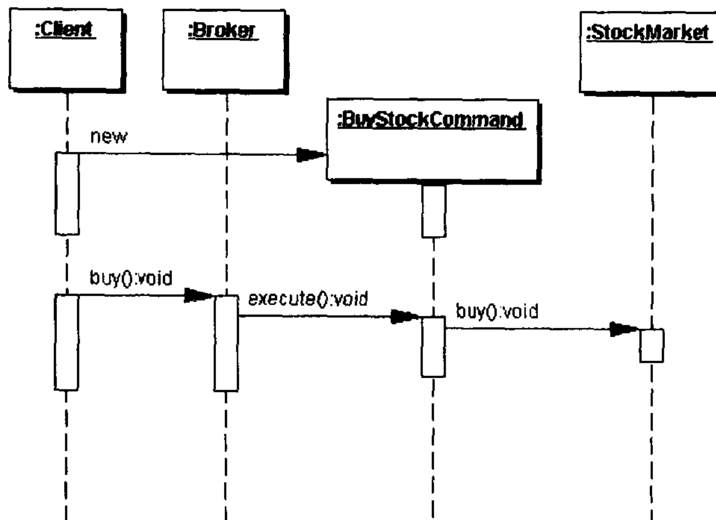
代码清单 32：具体命令类 SellStockCommand 的源代码

```
class SellStockCommand implements Command
{
    private StockMarket stock;

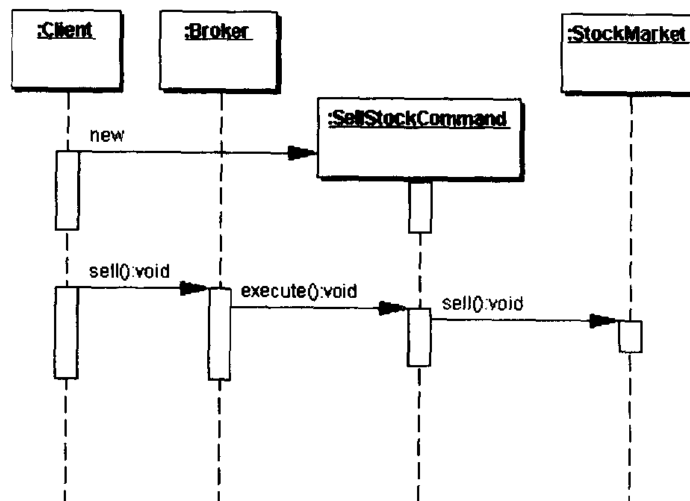
   /**
    *  构造子
    */
```

```
    public SellStockCommand ( StockMarket stock)
    {
        stock = stock;
    }
   /**
    *  执行方法
    */
    public void execute( )
    {
        stock.sell( );
    }
}
```

4. 股民通过股票经纪人买股票的时序图如下图所示。

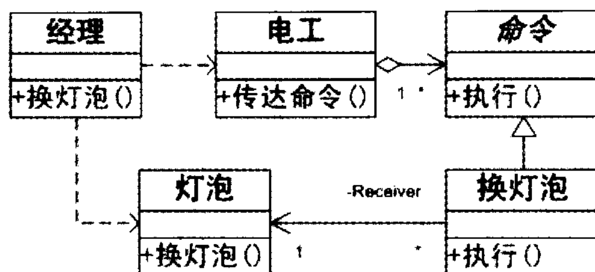


股民通过股票经纪人卖股票的时序图如下图所示。



5. 换电灯泡不需要经理亲自去。作为客户端经理只需要通过电工发出一个命令，命

令的接收者是那个新的电灯泡。命令一到，灯泡就会上去的。换灯泡的模拟系统设计图如下图所示。

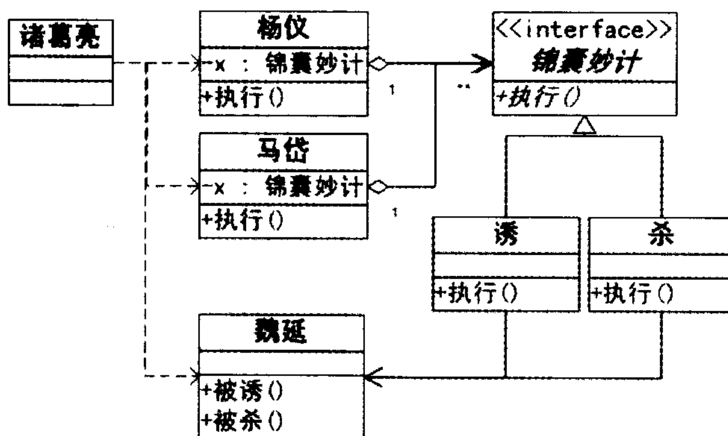


使用命令模式的优点是，命令及命令所代表的操作封装在命令对象里面。经理作为客户端只管发出命令，灯泡自己上不上去不是我的事。

6. Invoker 类（也就是电工）负责命令的执行，可以问一问他。

7. 笔者愚笨，依笔者看来，在“创世界”里软件工程处于第一位。圣经里说，上帝造光之前，“地是空虚混沌，渊面黑暗。神的灵运行在水面上。”这黑暗混沌描述的不正是软件工程吗？

8. 这个故事是命令模式。孔明密计杀魏延的系统设计图如下图所示。



显然，诸葛亮发出两个命令，即“诱”与“杀”，分别由两个 Invoker 对象即杨仪与马岱扮演请求者角色。命令的接收者当然是魏延自己。

因此，在历史上同样叫做“锦囊妙计”的故事，可能属于不同的模式，请参考“策略模式”一章。

第 48 章　专题：Swing 库中的命令撤销和恢复

在阅读本章之前，请首先阅读本书的“命令（Command）模式”一章。

48.1　在视窗系统中使用命令模式

在视窗应用程序里使用命令模式，可以有如下的特点：

（1）使命令所代表的操作与用户 GUI 界面分开。

（2）可以把相关的操作与单个的命令对象结合在一起。

（3）每一个按键都应当有它单独的类。

（4）由于命令对象需要与用户 GUI 界面发生关系，所以命令类要么是用户 GUI 界面类的内部类，要么是外部类。如果命令类是用户 GUI 界面类的内部类，那么命令类当然可以得到 GUI 界面类的私有属性。

如果命令类是独立的外部类，那么为使命令类得到 GUI 界面类的内部字段，可以有两种方法：第一是提供公开的方法，以便外界能够得到内部字段；第二是在通过命令类的构造子传入。第一种方法不好，它打破了封装的原则，把本该私有的变量向所有的外部类公开。第二种方法是本书所推荐的方法。

命令的撤销（undo）和恢复（redo）对很多的系统，特别是文字或图像文件系统来说，都是非常重要的。Swing 在 javax.swing.undo 库里提供了撤销和恢复的系统化处理方法。

javax.swing.undo 库虽然是 Swing 库的一部分，但它既可以与 Swing 构件一起使用，又可以与 AWT 构件一起使用，甚至可以在根本没有视窗构件的系统里使用。

48.2　Swing 的基本撤销功能

每一个命令或操作在 Swing 里都叫做一个编辑（Edit）。每一个可撤销的命令或操作在 Swing 里都叫做一个可撤销编辑（UndoableEdit），如下图所示。Swing 的撤销和恢复功能是通过下面的接口和类实现的。

（1）接口 UndoableEdit。一个类如果需要有撤销和恢复功能的话，就需要实现 UndoableEdit 接口。

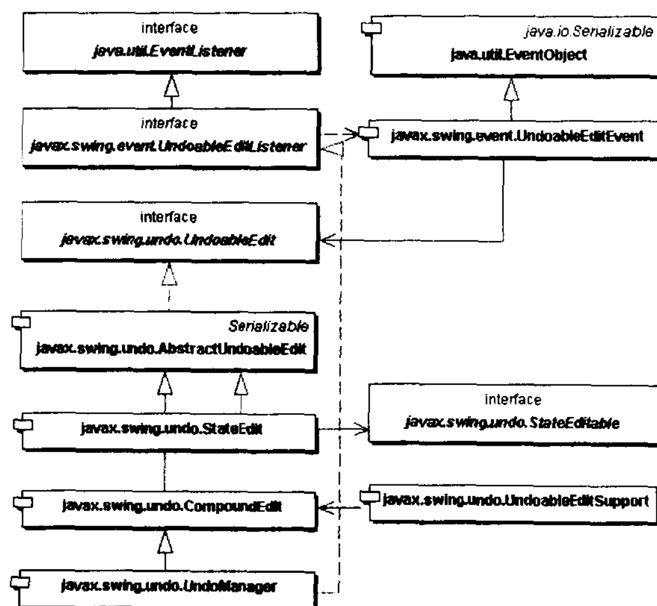
（2）抽象类 AbstractUndoableEdit 是对 UndoableEdit 接口的最小实现，也是

javax.swing.undo 库里其他类的基类。

（3）接口 StateEditable 是所有的可以编辑的类必须实现的接口。

（4）StateEdit 类代表一个可编辑类的状态。

（5）UndoManager 类负责管理所有的对一个可编辑类的编辑。



UndoableEdit 接口

这个接口的源代码，如代码清单 1 所示。本书把最重要的方法都给出了说明。

代码清单 1：UndoableEdit 接口的源代码

```
package javax.swing.undo;
import javax.swing.event.*;
public interface UndoableEdit {
    /**
     * 撤销编辑
     */
    public void undo() throws CannotUndoException;
    /**
     * 如果操作是可撤销的，返回 true；否则返回 false
     */
    public boolean canUndo();
    /**
     * 恢复一个已撤销的编辑
     */
    public void redo() throws CannotRedoException;
    /**
     * 如果一个操作是可以撤销的，返回 true；否则返回 false
     */
```

```
    public boolean canRedo();
    /**
     * 将一个编辑永久取消（而这个取消操作是不能撤销的）
     */
    public void die();
    /**
     * 将编辑 anEdit 吸收合并。如果吸收合并成功，返回 true；否则返回 false
     */
    public boolean addEdit(UndoableEdit anEdit);
    /**
     * 将当前的编辑替换为 anEdit
     */
    public boolean replaceEdit(UndoableEdit anEdit);
    public boolean isSignificant();
    public String getPresentationName();
    public String getUndoPresentationName();
    public String getRedoPresentationName();
}
```

可以看出，上面的接口规定了判断一个编辑是否可以撤销，以及将一个编辑撤销掉的方法。

AbstractUndoableEdit 类

如果能够重新设计这个类的话，本书会建议把它设计成一个抽象类。这个类不应当被直接实例化，被实例化的应当是它的子类，如代码清单 2 所示。

代码清单 2：AbstractUndoableEdit 类的源代码（节选）

```
package javax.swing.undo;
import javax.swing.UIManager;
import java.io.Serializable;
public class AbstractUndoableEdit implements UndoableEdit, Serializable
{
    /**
     * getUndoPresentationName()所返回的字符串
     */
    protected static final String UndoName = "Undo";
    /**
     * getRedoPresentationName()所返回的字符串
     */
    protected static final String RedoName = "Redo";
    /**
     * 默认值是 true。当此编辑被撤销后，此变量变为 false；恢复后又变为 true
     */
    boolean hasBeenDone;
    /**
```

```
 * 如果此编辑没有被取消，返回 true
 */
boolean alive;
public AbstractUndoableEdit()
{
 super();
 hasBeenDone = true;
 alive = true;
}
/**
 * 将一个编辑永久取消（而这个取消操作是不能撤销的）
 */
public void die()
{
    alive = false;
}
/**
 * 撤销编辑
 */
public void undo() throws CannotUndoException
{
    if (!canUndo())
    {
        throw new CannotUndoException();
    }
    hasBeenDone = false;
}
/**
 * 返还 true， 如果此编辑是活的，并且 hasBeenDone 是 true.
 */
public boolean canUndo()
{
 return alive && hasBeenDone;
}
/**
 * 恢复一个已撤销的编辑
 */
public void redo() throws CannotRedoException
{
 if (!canRedo())
 {
    throw new CannotRedoException();
 }
 hasBeenDone = true;
}
/**
```

```
     * 返还 true 如果此编辑是活的并且 hasBeenDone 是 false
     */
    public boolean canRedo()
    {       return alive && !hasBeenDone;       }
    /**
     * This default implementation returns false.
     */
    public boolean addEdit(UndoableEdit anEdit)
    {       return false;       }
    /**
     * This default implementation returns false.
     */
    public boolean replaceEdit(UndoableEdit anEdit)
    {       return false;       }
    /**
     * This default implementation returns true.
     */
    public boolean isSignificant()
    {       return true;       }
    public String getPresentationName()
    {   return "";   }
    public String getUndoPresentationName() {...}
    public String getRedoPresentationName() {...}
    public String toString(){...}
}
```

显然，这是一个缺省适配模式的应用。有了这个类作为一个 UndoableEdit 接口的最小实现，它的子类就可以省略掉不需要的方法。

关于缺省适配模式，请参见本书的“缺省适配（Default Adapter）模式”一章。

StateEditable 接口

这个接口要求两个方法，分别用来存储和取出一个状态，如代码清单 3 所示。

代码清单 3：StateEditable 接口的源代码（节选）

```
package javax.swing.undo;
import java.util.Hashtable;
public interface StateEditable
{
    /** 此类的 Resource ID */
    public static final String RCSID = ...;
    /**
     * 调用此方法将传入的状态存储到 state 里面。
     */
    public void storeState(Hashtable state);
    /**
```

```
     * 调用此方法从 state 取出状态。
     */
    public void restoreState(Hashtable state);
}
```

StateEdit 类

StateEdit 类扩展了 AbstractUndoableEdit，如代码清单 4 所示。

代码清单 4：StateEditable 接口的源代码（节选）

```
package javax.swing.undo;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
public class StateEdit extends AbstractUndoableEdit
{
    protected static final String RCSID = ...;
    //
    // 属性
    //
    /**
     * 被编辑的对象
     */
    protected StateEditable object;
    /**
     * 编辑发生前的信息存储在此处
     */
    protected Hashtable preState;
    /**
     * 在编辑发生后的状态信息存在此处
     */
    protected Hashtable postState;
    /**
     * undo/redo  的表象名
     */
    protected String undoRedoName;
    //
    // 构造子
    //
    /**
     * 创建并返还一个新的 StateEdit 对象。
     */
    public StateEdit(StateEditable anObject)
    {
        super();
        init (anObject,null);
```

```
}
public StateEdit(StateEditable anObject, String name)
{
    super();
 init (anObject,name);
}
protected void init (StateEditable anObject, String name)
{
 this.object = anObject;
 this.preState = new Hashtable(11);
 this.object.storeState(this.preState);
 this.postState = null;
 this.undoRedoName = name;
}
//
// 方法
//
/**
 * 得到 StateEditable 对象编辑后的状态并结束编辑
 */
public void end() {
 this.postState = new Hashtable(11);
 this.object.storeState(this.postState);
 this.removeRedundantState();
}
/**
 * 恢复编辑前的状态
 */
public void undo()
{
 super.undo();
 this.object.restoreState(preState);
}
/**
 * 将被编辑的对象恢复为编辑后的状态
 */
public void redo()
{
 super.redo();
 this.object.restoreState(postState);
}
/**
 * 返回此编辑的表象名
 */
public String getPresentationName()
{
```

```
        return this.undoRedoName;
    }
    //
    // 内部支持方法
    //
    protected void removeRedundantState() {...}
}
```

UndoManager 类

UndoManager 实现了 UndoableEditListener 接口，并扩展了 CompoundEdit 类，如代码清单 5 所示。

代码清单 5：UndoManager 类的源代码（节选）

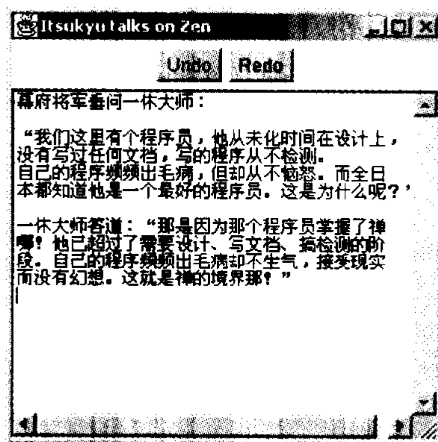
```
package javax.swing.undo;
import javax.swing.event.*;
import javax.swing.UIManager;
import java.util.*;
public class UndoManager extends CompoundEdit implements UndoableEditListener
{
    int indexOfNextAdd;
    int limit;
    public UndoManager() { ...}
    /**
     * 返还可以存储的总的编辑数目，默认值是 100。
     */
    public synchronized int getLimit()
    {    return limit;    }

    /**
     * 将所存储的所有编辑清掉
     */
    public synchronized void discardAllEdits() {...}
    protected void trimForLimit() {...}
    protected void trimEdits(int from, int to) {...}
    /**
     * 设定所能存储的编辑的数目，默认值是 100。
     */
    public synchronized void setLimit(int l) {...}
    protected UndoableEdit editToBeUndone() {...}
    protected UndoableEdit editToBeRedone() {...}
    protected void undoTo(UndoableEdit edit) throws CannotUndoException {...}
    protected void redoTo(UndoableEdit edit) throws CannotRedoException {...}
    /**
     * Undo 或者 redo ，哪一个合适使用哪一个
     */
```

```
    public synchronized void undoOrRedo()
     throws CannotRedoException, CannotUndoException {...}
    /**
     * 是否可以 undo 或者 redo
     */
    public synchronized boolean canUndoOrRedo() {...}
    public synchronized void undo() throws CannotUndoException {...}
    public synchronized boolean canUndo() {...}
    public synchronized void redo() throws CannotRedoException {...}
    public synchronized boolean canRedo() {...}
    public synchronized boolean addEdit(UndoableEdit anEdit) {...}
    public synchronized void end() {...}
    public synchronized String getUndoOrRedoPresentationName() {...}
    public synchronized String getUndoPresentationName() {...}
    public synchronized String getRedoPresentationName() {...}
    public void undoableEditHappened(UndoableEditEvent e) {...}
    public String toString() {...}
}
```

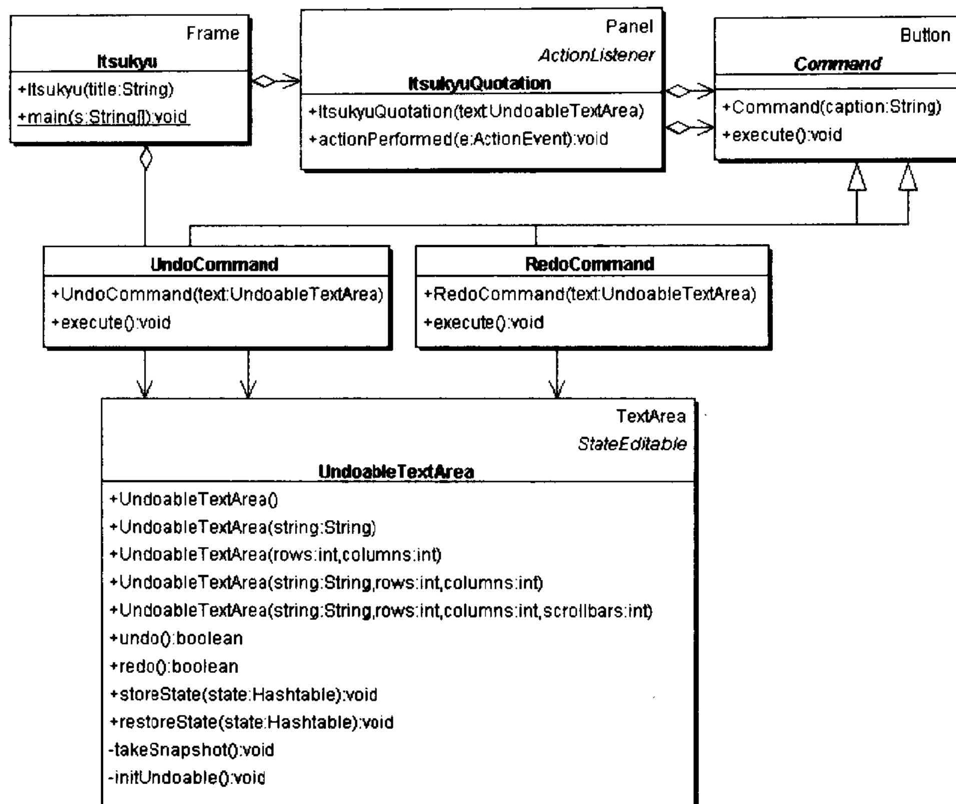
48.3　一休论禅的文字框

一休（Itsukyu），全名一休宗纯，史称狂僧，是日本僧侣史上最有名的三位和尚之一。这里提供一个一休大师可以论禅的文字框程序，如下图所示。



为让狂僧一休可以自由论禅，本章允许一休可以随便修改他说过的话，换言之，一休的文字框提供无穷次的撤销（undo）和恢复（redo）。

系统的设计结构如下图所示。可以看出，这个系统由一个视窗类 Itsukyu、一个可撤销的文字框 UndoableTextArea、抽象命令角色 Command，以及具体命令角色 UndoCommand 和 RedoCommand 等组成。



正如在前面所指出的，javax.swing.undo 库不一定非要与 Swing 视窗构件一起使用不可。在一休系统里，所有的视窗组件都是用 AWT 组件而非 Swing 组件，虽然使用的撤销和恢复功能来自 Swing。

视窗类 Itsukyu 的源代码如代码清单 6 所示。

代码清单 6：一休（Itsukyu）类的源代码

```
import java.util.Hashtable;
import java.awt.*;
import java.awt.event.*;
public class Itsukyu extends Frame
{
    private static UndoableTextArea text ;
    static ItsukyuQuotation panel;
    public Itsukyu(String title)
    {
        super(title);
    }
    public static void main(String[] s)
    {
        // create an undoable text area
        text = new UndoableTextArea("Your text here.");
        // Create app panel
        panel = new ItsukyuQuotation(text);
        // Create a frame for app
```

```
            Itsukyu frame = new Itsukyu("Itsukyu talks on Zen");
            // Add a window listener for window close events
            frame.addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent e) { System.exit(0);}
            } );
            // Add app panel to content pane
            frame.add(panel);
            // Set initial frame size and make visible
            frame.setSize (300, 300);
            frame.setVisible(true);
        }
    }
```

下面是视窗上的 Panel 类的源代码，如代码清单 7 所示。可以看出，这个类同时实现了 AWT 的事件监听接口 ActionListener。

代码清单 7：一休论坛类 ItsukyuQuotation 的源代码

```
import java.util.Hashtable;
import java.awt.*;
import java.awt.event.*;
import javax.swing.undo.*;
class ItsukyuQuotation extends Panel implements ActionListener
{
    private final Command undo ;
    private final Command redo ;
    // Constructor
    public ItsukyuQuotation(UndoableTextArea text)
    {
        setLayout(new BorderLayout());
        // create a toolbar for buttons
        Panel toolbar = new Panel();
        // create undo and redo buttons
        // add listeners for buttons
        undo = new UndoCommand(text);
        redo = new RedoCommand(text);
        undo.addActionListener (this);
        redo.addActionListener (this);
        // add buttons to toolbar
        toolbar.add(undo);
        toolbar.add(redo);
        // add toolbar and text area to panel
        add(toolbar, "North");
        add(text, "Center");
    }
    //---------------------------------------
    public void actionPerformed(ActionEvent e)
```

```
        {
            Command cmd = (Command)e.getSource();
            cmd.execute();
        }
    }
```

抽象命令类 Command 的源代码如代码清单 8 所示。

代码清单 8：抽象命令类的源代码

```
import java.awt.Button;
public abstract class Command extends Button
{
    public Command(String caption)
    {
        super(caption);
    }
    public abstract void execute();
}
```

下面的撤销命令类是抽象命令类的子类，这个类代表的是对编辑的撤销，如代码清单 9 所示。

代码清单 9：撤销命令类的源代码

```
import javax.swing.*;
public class UndoCommand extends Command
{
    UndoableTextArea text;
    public UndoCommand(UndoableTextArea text)
    {
        super("Undo");
        this.text = text;
    }
    public void execute()
    {
        text.undo();
    }
}
```

下面是恢复命令类的源代码，如代码清单 10 所示。这个类代表的是对编辑的恢复，一个编辑在被撤销后，可以通过这个恢复命令加以恢复。

代码清单 10：恢复命令的源代码

```
import javax.swing.*;
public class RedoCommand extends Command
{
    UndoableTextArea text;
    public RedoCommand(UndoableTextArea text)
```

```
    {
        super("Redo");
        this.text = text;
    }
    public void execute()
    {
        text.redo();
    }
}
```

好了，下面是这个系统的关键部分，也就是可以撤销的文字框类 UndoableTextArea 的源代码，如代码清单 11 所示。

代码清单 11：UndoableTextArea 类的源代码

```
import java.util.Hashtable;
import java.awt.*;
import java.awt.event.*;
import javax.swing.undo.*;
/**
 *   UndoableTextArea: An undoable extension of TextArea
 */
class UndoableTextArea extends TextArea implements StateEditable
{
    private final static String KEY_STATE = "UndoableTextAreaKey";   // hash key
    private boolean textChanged = false;
    private UndoManager undoManager;
    private StateEdit currentEdit;
    public UndoableTextArea()
    {
        super();
        initUndoable();
    }
    public UndoableTextArea(String string)
    {
        super(string);
        initUndoable();
    }
    public UndoableTextArea(int rows, int columns)
    {
        super(rows, columns);
        initUndoable();
    }
    public UndoableTextArea(String string, int rows, int columns)
    {
        super(string, rows, columns);
        initUndoable();
    }
```

```
public UndoableTextArea(String string, int rows, int columns, int scrollbars)
{
    super(string, rows, columns, scrollbars);
    initUndoable();
}
// method to undo last edit
public boolean undo()
{
    try
    {
        undoManager.undo();
        return true;
    }
    catch (CannotUndoException e)
    {
        System.out.println("Can't undo");
        return false;
    }
}
// method to redo last edit
public boolean redo()
{
    try
    {
        undoManager.redo();
        return true;
    }
    catch (CannotRedoException e)
    {
        System.out.println("Can't redo");
        return false;
    }
}
// Implementation of StateEditable interface
//
// save and restore data to/from hashtable
public void storeState(Hashtable state)
{
    state.put(KEY_STATE, getText());
}
public void restoreState(Hashtable state)
{
    Object data = state.get(KEY_STATE);
    if (data != null)
    {
        setText((String)data);
```

```
    }
}
// Snapshots current edit state
private void takeSnapshot()
{
    // snapshot only if text changed
    if (textChanged)
    {
        // end current edit and notify undo manager
        currentEdit.end();
        undoManager.addEdit(currentEdit);
        // reset text changed semaphore and create a new current edit
        textChanged = false;
        currentEdit = new StateEdit(this);
    }
}
// Helper method to initialize object to be undoable
private void initUndoable ()
{
    // create an undo manager to manage undo operations
    undoManager = new UndoManager();
    // create a StateEdit object to represent the current edit
    currentEdit = new StateEdit(this);
    // add listeners for various edit-related events
    // use these events to determine when to snapshot current edit
    // key listener looks for action keys (non-character keys)
    addKeyListener(new KeyAdapter()
    {
        public void keyPressed(KeyEvent event)
        {
            if (event.isActionKey())
            {
                // snapshot on any action keys
                takeSnapshot();
            }
        }
    });
    // focus listener looks for loss of focus
    addFocusListener(new FocusAdapter()
    {
        public void focusLost(FocusEvent event)
        {
            // snapshot when control loses focus
            takeSnapshot();
        }
    });
```




```
            // text listener looks for text changes
            addTextListener(new TextListener()
            {
                public void textValueChanged(TextEvent event)
                {
                    textChanged = true;

                    // snapshot on every change to text
                    // might be too granular to be practical
                    //
                    // this will shapshot every keystroke when typing
                    takeSnapshot();
                }
            } );
        }
    }
```

可以看到，UndoableTextArea 继承自 java.awt.TextArea，实现了 StateEditable 接口。因此，UndoableTextArea 类需要实现 StateEditable 接口所要求的方法。storeState()及restoreState()方法分别负责存储一个新的状态和恢复一个状态。

参考文献

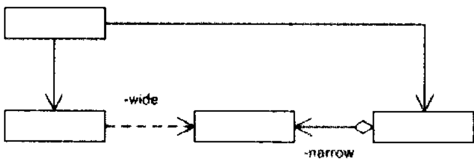
[FOLEY98] Michael W. Foley and Mark McCulley. JFC Unleashed. published by Macmillan Computer Publishing，1998

第 49 章　备忘录（Memento）模式

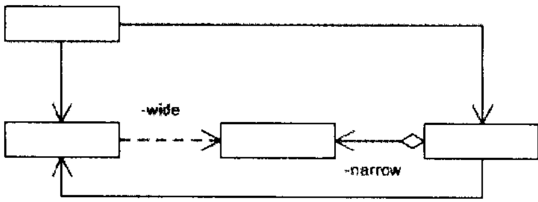
备忘录模式（Memento Pattern）又叫做快照模式（Snapshot Pattern）或 Token 模式，是对象的行为模式[GOF95]。

备忘录（Memento）对象是一个用来存储另外一个对象内部状态的快照（snapshot）的对象。备忘录模式（Memento Pattern）的用意是在不破坏封装（encapsulation）的条件下，将一个对象的状态捕捉（Capture）住，并外部化（Externalize），存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。备忘录模式常常与命令模式和迭代子模式一同使用。

根据角色责任划分上面的细微区别，类图略有不同，备忘录模式的简略类图如下图所示。



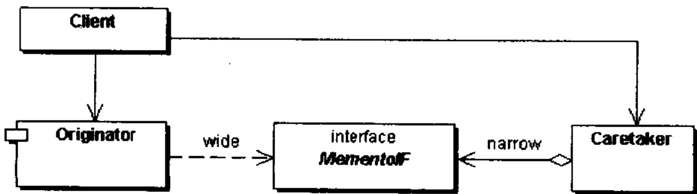
使用了增强的负责人（Caretaker）角色的备忘录模式的简略类图如下图所示。



常见的软件系统往往不止存储一个状态，而是需要存储多个状态。这些状态常常是一个对象历史发展的不同阶段的快照，存储这些快照的备忘录对象叫做此对象的历史；某一个快照所处的位置叫做检查点（Check Point）。

49.1　备忘录模式的结构

备忘录模式的结构图如下图所示。



备忘录模式所涉及的角色有三个：备忘录（Memento）角色、发起人（Originator）角色和负责人（Caretaker）角色。

备忘录（Memento）角色

备忘录角色有如下的责任。

（1）将发起人（Originator）对象的内部状态存储起来。备忘录可以根据发起人对象的判断来决定存储多少发起人（Originator）对象的内部状态。

（2）备忘录可以保护其内容不被发起人（Originator）对象之外的任何对象所读取。备忘录有两个等效的接口：

- 窄接口：负责人（Caretaker）对象（和其他除发起人对象之外的任何对象）看到的是备忘录的窄接口（narrow interface），这个窄接口只允许它把备忘录对象传给其他的对象；
- 宽接口：与负责人对象看到的窄接口相反的是，发起人对象可以看到一个宽接口（wide interface），这个宽接口允许它读取所有的数据，以便根据这些数据恢复这个发起人对象的内部状态。

发起人（Originator）角色

发起人角色有如下的责任：

- 创建一个含有当前的内部状态的备忘录对象。
- 使用备忘录对象存储其内部状态。

负责人（Caretaker）角色

负责人角色有如下的责任：

- 负责保存备忘录对象。
- 不检查备忘录对象的内容。

49.2 备忘录模式的白箱实现

宽接口和白箱

在 Java 语言中实现备忘录模式时，实现“宽”和“窄”两个接口并不是容易的事。如果暂时忽略两个接口的区别，仅为备忘录角色提供一个宽接口的话，情况就变得简单得多。但是由于备忘录角色对任何对象都提供一个接口，即宽接口，备忘录角色的内部所存储的状态就对所有对象公开。因此这个实现又叫做“白箱实现”。

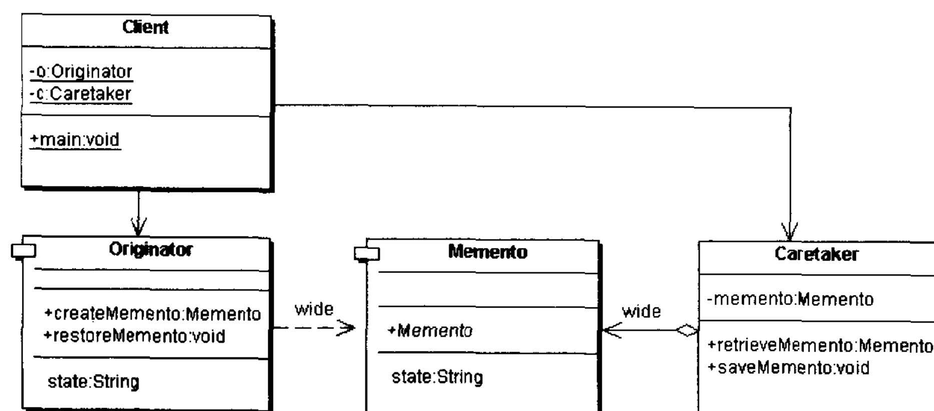
“白箱”实现将发起人角色的状态存储在一个大家都看得到的地方，因此是破坏封装

性的。但是通过程序员自律，同样可以在一定程度上实现模式的大部分用意。因此白箱实现仍然是有意义的。

备忘录模式的“白箱”实现又叫做“白箱”备忘录模式。

“白箱”备忘录模式的实现

如下图所示，其中给出一个示意性的“白箱实现”，读者可以将这个例子当做一个框架，在将它应用到自己的系统中去的时候再补充必要的细节。



下面给出系统的源代码。首先是发起人角色的源代码，可以看出，发起人角色利用一个新创建的备忘录对象将自己的内部状态存储起来，如代码清单 1 所示。

代码清单 1：发起人角色的源代码

```
package com.javapatterns.memento.whitebox;
public class Originator
{
    private String state;
   /**
    *    工厂方法，返还一个新的备忘录对象
    */
    public Memento createMemento()
    {
        return new Memento(state);
    }
   /**
    *    将发起人恢复到备忘录对象所记载的状态
    */
    public void restoreMemento(Memento memento)
    {
        this.state = memento.getState();
    }
   /**
    *    状态的取值方法
    */
```

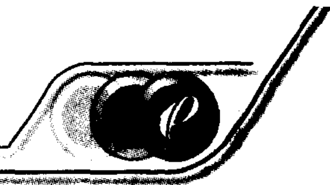
```
    public String getState()
    {
        return this.state;
    }
   /**
    *    状态的赋值方法
    */
    public void setState(String state)
    {
        this.state = state;
        System.out.println("Current state = " + this.state);
    }
}
```

下面是备忘录角色的源代码，如代码清单 2 所示。备忘录对象将发起人对象传入的状态存储起来。备忘录角色会根据发起人对象的判断来决定将发起人对象的内部状态的多少存储起来。

备忘录模式要求备忘录对象提供两个不同的接口：一个宽接口提供给发起人对象，另一个窄接口提供给所有其他的对象，包括负责人对象。宽接口允许发起人读取到所有的数据；窄接口只允许它把备忘录对象传给其他的对象而看不到内部的数据。这里所给出的示意性实现没有给出两个不同的接口，关于这一点的讨论请见模式的实现一节。

代码清单 2：备忘录角色的源代码

```
package com.javapatterns.memento.whitebox;
public class Memento
{
    private String state;
   /**
    *    构造子
    */
    public Memento(String state)
    {
        this.state = state;
    }
   /**
    *    状态的取值方法
    */
    public String getState()
    {
        return this.state;
    }
   /**
    *    状态的赋值方法
    */
    public void setState(String state)
    {
```



```
        this.state = state;
    }
}
```

下面是负责人角色的源代码，如代码清单 3 所示。负责人角色负责保存备忘录对象，但是从不修改（甚至不查看）备忘录对象的内容（一个更好的实现是负责人对象根本无法从备忘录对象中读取和修改其内容，这一点也请见后面模式的实现一节）。

代码清单 3：负责人角色的源代码

```
package com.javapatterns.memento.whitebox;
public class Caretaker
{
    private Memento memento;
   /**
    *     备忘录的取值方法
    */
    public Memento retrieveMemento()
    {
        return this.memento;
    }
   /**
    *     备忘录的赋值方法
    */
    public void saveMemento(Memento memento)
    {
        this.memento = memento;
    }
}
```

下面给出一个示意性的客户端的源代码，如代码清单 4 所示。

代码清单 4：客户端角色的源代码

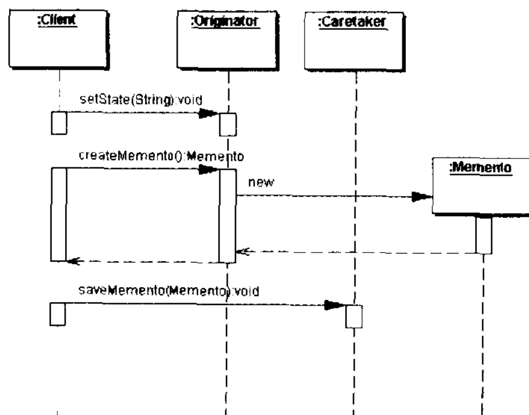
```
package com.javapatterns.memento.whitebox;
public class Client
{
    private static Originator o = new Originator();
    private static Caretaker c = new Caretaker();
    public static void main(String[] args)
    {
        // 改变负责人对象的状态
        o.setState("On");
        // 创建备忘录对象，并将发起人对象的状态存储起来
        c.saveMemento( o.createMemento() );
        // 修改发起人对象的状态
        o.setState("Off");
        // 恢复发起人对象的状态
        o.restoreMemento( c.retrieveMemento() );
```

```
    }
}
```

在上面的这个示意性的客户端角色里面，首先将发起人对象的状态设置成“On”（或者任何有效状态），并且创建一个备忘录对象将这个状态存储起来；然后将发起人对象的状态改成“Off”（或者任何有效状态）；最后又将发起人对象恢复到备忘录对象所存储起来的状态，即“On”状态（或者先前所存储的任何状态）。

“白箱”实现的活动时序

系统的时序图更能够反映出系统内各个角色被调用的时间顺序。如下图所示是将发起人对象的状态存储到白箱备忘录对象中去的时序图。



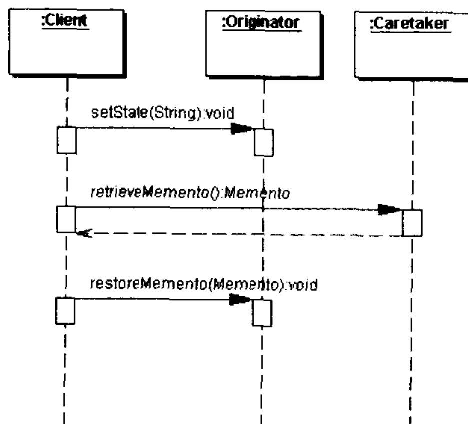
可以看出，将发起人对象的状态存储到备忘录中时，系统运行的时序是这样的：

（1）将发起人对象的状态设置成“On”。

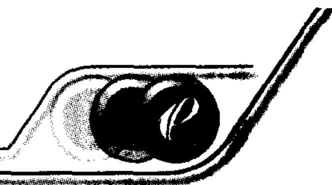
（2）调用发起人角色的 createMemento()方法，创建一个备忘录对象将这个状态存储起来。

（3）将备忘录对象存储到负责人对象中去。

将发起人对象恢复到备忘录对象所记录的状态的时序图如下图所示。



可以看出，将发起人对象恢复到备忘录对象所记录的状态时，系统运行的时序是这样



的：

（1）将发起人对象的状态设置成“Off”；

（2）将备忘录对象从负责人对象中取出；

（3）将发起人对象恢复到备忘录对象所存储起来的状态，即“On”状态。

白箱实现的优缺点

白箱实现的一个明显的好处是比较简单，因此常常用做教学目的。白箱实现的一个明显的缺点是破坏对发起人状态的封装。



在上面给出的示意性实现中只能存储一个状态（又叫做一个检查点），本章会在后面的小节中给出一个能够存储多个状态的设计。

49.3　双重接口及其在 Java 语言中的实现

宽接口和窄接口

在前面对备忘录模式的描述里，“不破坏封装”是一个关键的修饰词。一个系统可能需要将某个对象的状态，比如一个可视构件对象的位置、颜色、大小等状态参数全部或部分地存储起来，以便将来可以恢复到此状态。通常情况下，一个对象不应当将自己的内部状态暴露给外界，这就是说，如果此对象提供一些公开接口以提供其内部状态的话，会产生外部代码直接利用这些接口修改此对象的潜在危险。

备忘录模式利用宽接口（wide interface）和窄接口（narrow interface）的设计解决了这个问题。这种在接口上的区别对于 C++语言而言，意味着将 Memento 的接口设成私有，同时将 Originator 设为 Memento 的友类，如代码清单 5 所示。

代码清单 5：如何在 C++中实现两个接口

```
Class Memento {
public:
    virtual ~Memento();
private:
    friend class Originator;
    Memento();
    void setState(State*);
    State* GetState();
    ...
```

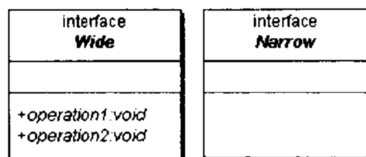
所谓的双重接口，就是对某一个对象提供宽接口，而对另一些对象提供窄接口。根据编程语言的性能，双重接口的实现方式有所不同。

宽接口和窄接口在 Java 语言中的实现

现在考虑一下，怎样才能在 Java 语言中实现双重接口。

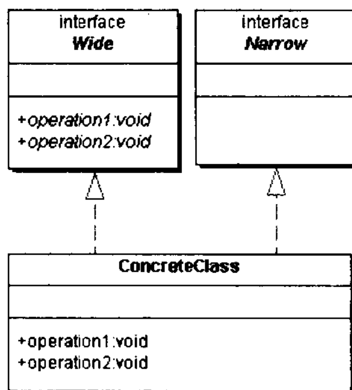
两个接口

首先，既然谈到两个接口，那么就不妨从两个 Java 接口开始吧。下图所示就是两个 Java 接口，一个叫做 Wide，一个叫做 Narrow。Wide 接口有两个方法，分别是 operation1()和 operation2()；而 Narrow 则是一个窄的不能再窄的接口：它根本没有声明任何方法。读过本书“Java 语言的接口”一章的读者会看出，Narrow 是一个标识接口。



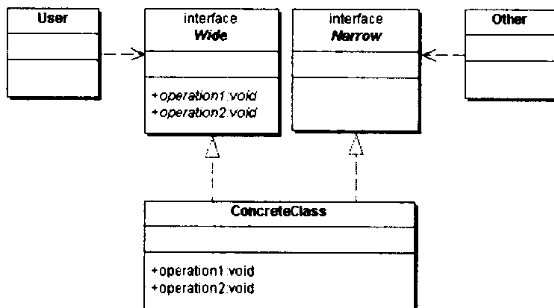
一个具体类的双重接口

既然要谈到一个具体类，那么就创建这样一个具体类，叫 ConcreteClass 吧。这个类同时实现两个接口，Wide 和 Narrow，如下图所示。

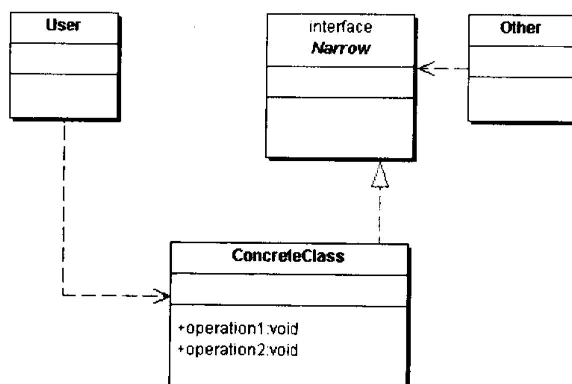


对不同对象提供不同的接口

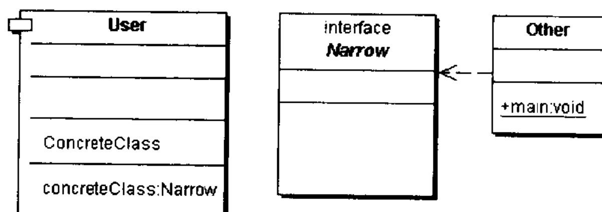
现在必须考虑核心的问题了，具体类 ConcreteClass 必须向所有的对象提供 Narrow 接口，而对一个特殊的对象提供 Wide 接口。假设有一个 User 类和一个 Other 类，Other 类代表所有可能的客户端，而 User 类代表那个特殊的类型，如下图所示。



当然，Wide 接口可以省略，它所扮演的角色可以由 ConcreteClass 自己扮演。在下面仍然会提到宽接口，但是所指的不再是名为 Wide 的 Java 接口，而仅仅是 ConcreteClass 自己所声明的接口而已，如下图所示。



这样能阻止除了 User 对象之外的所有其他对象引用宽接口吗？不能。那么有没有什么办法使得宽接口只对 User 对象存在呢？将宽接口（也就是 ConcreteClass 本身）设计成为 User 类的内部成员类可不可以呢？如下图所示。



这时，有经验的读者马上就会看出问题：ConcreteClass 虽然是 User 的内部成员类，但是它仍然不仅对 User 类存在，而且对所有的同一个 package 中的类都存在。换言之，成员类并没有私有访问权限。但是，如果将 ConcreteClass 的所有方法都设计成私有的，不就达到目的了吗？是的，这就是所要寻找的解决方案。

User 类的源代码如代码清单 6 所示。

代码清单 6：User 类含有 ConcreteClass 作为内部成员类

```
public class User
{
    // 下面是内部成员类
    class ConcreteClass implements Narrow
    {
        private void operation1()
        {
            System.out.println("operation1()");
        }
        private void operation2()
        {
            System.out.println("operation2()");
        }
```

```
    }
    // 下面这个方法向外界提供成员类的实例
    public Narrow getConcreteClass()
    {
        return (Narrow) new ConcreteClass();
    }
}
```

这就是说，只有 User 对象可以得到并向外界提供 ConcreteClass 的实例。User 对象可以调用 ConcreteClass 对象的私有方法，因为 ConcreteClass 是 User 的内部类。当 Other 对象收到这个实例时，可以根据 Narrow 类型收到，但是这个 Narrow 接口没有提供任何方法；也可以根据 ConcreteClass 类型收到，但是 ConcreteClass 类型只有私有方法，因此 Other 对象无法得到 ConcreteClass 对象的内部信息。

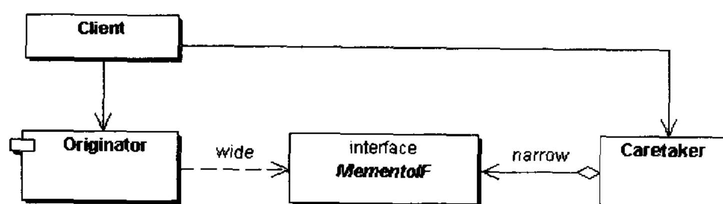
这就给出了一个 Java 语言中的双接口设计方案。

对 Java 语言基础知识了解不够的读者，读上面的分析可能有困难。如果是这样的话，请参见本章最后的习题，或者阅读一本 Java 语言的基础读物。

49.4 备忘录模式的黑箱实现

将解决方案应用到备忘录模式的实现

对于 Java 语言而言，可以将 Memento 设成 Originator 类的内部类，从而将 Memento 对象封装在 Originator 里面；在外部提供一个标识接口 MementoIF 给 Caretaker 以及其他对象。这样，Originator 类看到的是 Memento 的所有接口，而 Caretaker 以及其他对象看到的仅仅是标识接口 MementoIF 所暴露出来的接口。使用内部类实现备忘录模式的简略类图如下图所示。

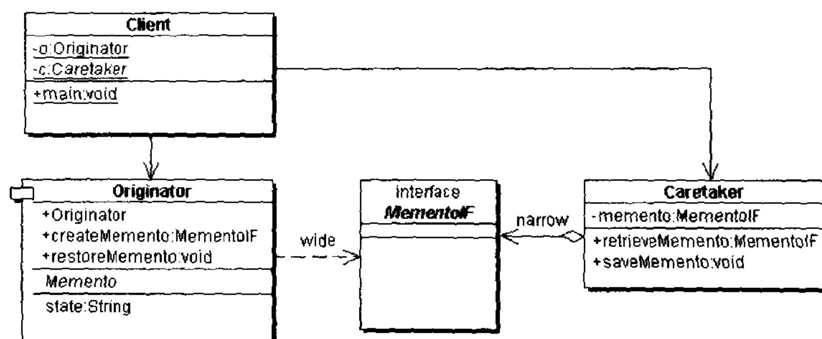


这一做法最早是由文献[GRAND98]提出来的。为了便于读者掌握设计的细节，下面就给出一个具体化的实现：

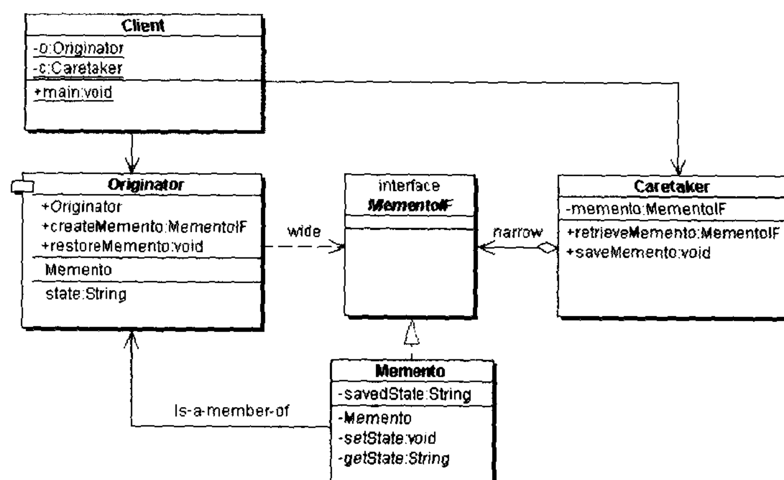
黑箱设计

从黑箱实现的具体设计类图可以看出，Originator 类有一个保护成员类 Memmento。这个 Memmento 类实现了 MementoIF 接口。由于 Memmento 类是仅对 Originator 类才存在

的，所有的其他对象都可以得到 Memmento 类的实例，但只能得到它的 MementoIF 接口，这就巧妙地实现了双重接口。使用内部类实现备忘录模式的类图如下图所示。



如果需要将内部类明显表示出来，那么上面的类图可以重新绘制，如下图所示。



下面给出的就是改造之后的发起人角色的源代码，如代码清单 7 所示。

代码清单 7：Originator 角色的源代码

```
package com.javapatterns.memento.blackbox;
public class Originator
{
    private String state;
  /**
   *     构造子
   */
    public Originator()
    {
    }
  /**
   *     工厂方法，返还一个新的备忘录对象
   */
    public MementoIF createMemento()
    {
```

```
        return new Memento( this.state );
    }

    /**
     *    将发起人恢复到备忘录对象记录的状态
     */
    public void restoreMemento( MementoIF memento)
    {
        Memento aMemento = (Memento) memento;
        this.setState( aMemento.getState() );
    }
    /**
     *    状态的取值方法
     */
    public String getState()
    {
        return this.state;
    }
    /**
     *    状态的赋值方法
     */
    public void setState(String state)
    {
        this.state = state;
        System.out.println("state = " + state);
    }

    /**
     *    内部成员类，备忘录
     */
    protected class Memento implements MementoIF
    {
        private String savedState;

        /**
         *    构造子
         */
        private Memento(String someState)
        {
            savedState = someState;
        }

        /**
         *    状态的赋值方法
         */
        private void setState(String someState)
```

```
            {
                savedState = someState;
            }
            /**
             *    状态的取值方法
             */
            private String getState()
            {
                return savedState;
            }
        }
    }
```

可以看到，在上面的 Originator 类中定义了一个内部的 Memento 类。由于此 Memento 类的全部接口都是私有的，因此只有它自己和发起人（Originator）类可以调用。

下面给出的是窄接口 MementoIF，这是一个标识接口，因为它没有定义出任何的方法，如代码清单 8 所示。

代码清单 8：MementoIF 角色是一个标识接口

```
package com.javapatterns.memento.blackbox;
public interface MementoIF
{
}
```

负责人角色的扮演者 Caretaker 类的源代码如代码清单 9 所示。读者可以看到，负责人角色能够得到的备忘录对象是以 MementoIF 为接口的，由于这个接口仅仅是一个标识接口，因此负责人角色不可能改变这个备忘录对象的内容。

代码清单 9：Caretaker 角色的源代码

```
package com.javapatterns.memento.blackbox;
public class Caretaker
{
    private MementoIF memento;
    /**
     *    备忘录的取值方法
     */
    public MementoIF retrieveMemento()
    {
        return this.memento;
    }
    /**
     *    备忘录的赋值方法
     */
    public void saveMemento(MementoIF memento)
    {
        this.memento = memento;
```

```
    }
}
```

客户端的源代码如代码清单 10 所示。

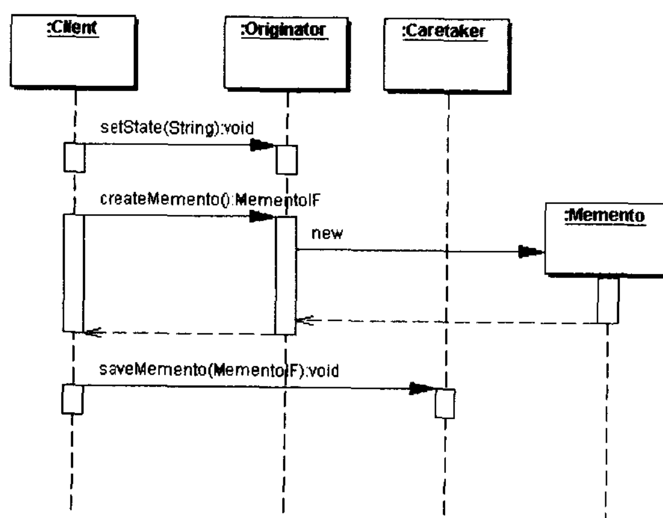
代码清单 10：客户端角色的源代码

```
package com.javapatterns.memento.blackbox;
public class Client
{
    private static Originator o = new Originator();
    private static Caretaker c = new Caretaker();
    public static void main(String[] args)
    {
        // 改变负责人对象的状态
        o.setState("On");
        // 创建备忘录对象，并将发起人对象的状态存储起来
        c.saveMemento( o.createMemento() );

        // 修改发起人对象的状态
        o.setState("Off");
        // 恢复发起人对象的状态
        o.restoreMemento( c.retrieveMemento() );
    }
}
```

黑箱实现的时序

系统的时序图更能够反映出系统内各个角色被调用的时间顺序。如下图所示，便是将发起人对象的状态存储到黑箱备忘录对象中去的时序图。



可以看出，将发起人对象的状态存储到备忘录中时，系统运行的时序是这样的：

（1）将发起人对象的状态设置成“On”。

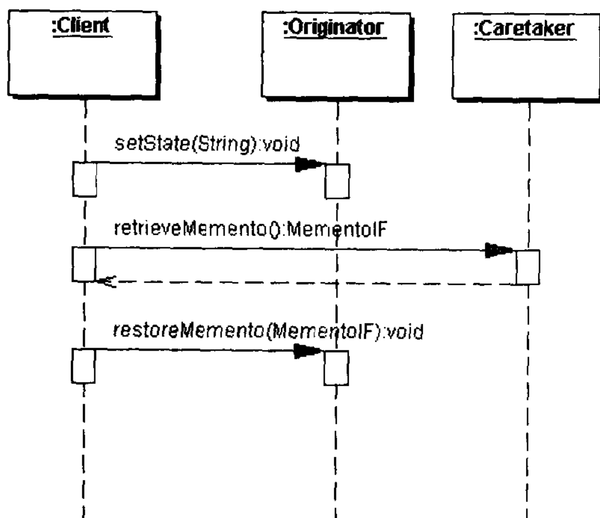


（2）调用发起人角色的 createMemento()方法，创建一个备忘录对象将这个状态存储起来。


此时 createMemento()方法返还的其实是明显类型为 MementoIF 类型，真实类型为 Originator 内部的 Memento 类型的对象。

（3）将备忘录对象存储到负责人对象中去。由于负责人对象拿到的仅是 MementoIF 类型，因此无法读出备忘录对象内部的状态。

将发起人对象恢复到备忘录对象所记录的状态的时序图，如下图所示。



将发起人对象恢复到备忘录对象所记录的状态时，系统运行的时序是这样的：

（1）将发起人对象的状态设置成“Off”；

（2）将备忘录对象从负责人对象中取出。注意此时仅能得到 MementoIF 接口，因此无法读出此对象的内部状态；

（3）将发起人对象的状态恢复成备忘录对象所存储起来的状态，即“On”状态。由于发起人对象的内部类 Memento 实现了 MementoIF 接口，这个内部类是传入的备忘录对象的真实类型，因此发起人对象可以利用内部类 Memento 的私有接口读出此对象的内部状态。

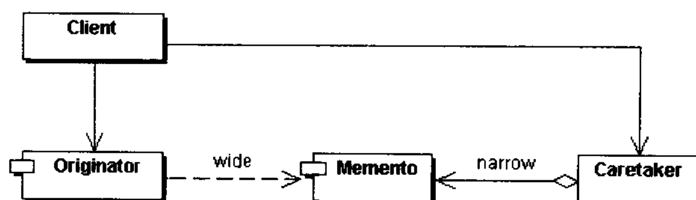

在上面给出的示意性实现中只能存储一个状态，或者叫一个检查点（Check Point）。在后面会给出一个能够存储多个检查点的实现。

49.5　负责人角色的增强

在前面的两个实现中，负责人角色所承担的责任并不包括操控发起人角色进行备忘录的创建和状态的恢复，这两个责任是由客户端角色直接调用发起人角色和备忘录角色做到的。

如果能让负责人角色调用备忘录角色和发起人角色，进行备忘录创建和根据备忘录恢复发起人状态，那么客户端便不再需要协调备忘录角色和发起人角色，而只需要调用负责人角色即可。要做到这一点，负责人角色就必须持有一个对发起人角色的引用。

如下图所示，此模式类图就体现了具有增强功能的负责人角色，其中从负责人角色到发起人角色的引用就代表了对发起人角色的操控。



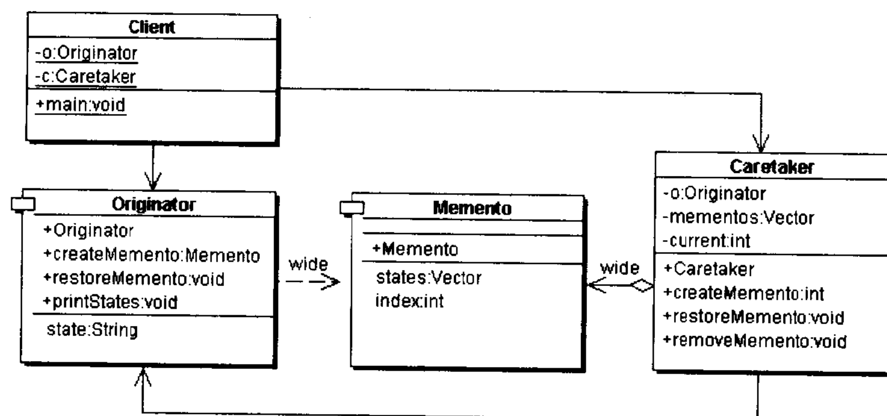
为了节省正文的篇幅，本书将前面的白箱实现中的负责人角色的增强化移到问答题中给出。此外，下面的多重检查点的设计也体现了这种对负责人角色进行增强的做法，读者也可以参考下面的实现。

49.6 备忘录模式与多重检查点

前面所给出的白箱和黑箱的示意性实现都是只存储一个状态的简单实现，也可以叫做只有一个检查点。常见的软件系统往往需要存储不止一个状态，而是需要存储多个状态，或者叫做有多个检查点。

设计图

备忘录模式可以将发起人对象的状态存储到备忘录对象里面，备忘录模式可以将发起人对象恢复到备忘录对象所存储的某一个检查点上。下图所示的设计就给出了一个示意性的、有多重检查点的备忘录模式的实现。



有些备忘录模式的实现在理论上允许有无限个状态，有些实现只允许有有限个状态。这里给出的例子在理论上允许有无限个状态。

源代码

下面给出这个实现的源代码。首先是发起人角色的源代码，如代码清单 11 所示。显然发起人角色的状态是存在于一个 Vector 对象中的，每一个状态都有一个指数 index，叫做检查点指数。此外 printStates()方法是一个提供辅助功能的方法。

代码清单 11：发起人角色的源代码

```
package com.javapatterns.memento.whiteboxcheckpoints;
import java.util.Vector;
import java.util.Enumeration;
public class Originator
{
    private Vector states;
    private int index;
   /**
    *    构造子
    */
    public Originator()
    {
        states = new Vector();
        index = 0;
    }
   /**
    *    工厂方法，返还一个新的备忘录对象
    */
    public Memento createMemento()
    {
        return new Memento(states, index);
    }
   /**
    *    将发起人恢复到备忘录对象记录的状态上
    */
    public void restoreMemento(Memento memento)
    {
        states = memento.getStates();
        index = memento.getIndex();
    }
   /**
    *    状态的赋值方法
    */
    public void setState(String state)
    {
        this.states.addElement(state);
        index++;
    }
```




```
    /**
     *    辅助方法，打印出所有的状态
     */
     public void printStates()
     {
          System.out.println("Total number of states : "
               + index);
          for(Enumeration e = states.elements();
               e.hasMoreElements(); )
          {
               System.out.println( e.nextElement() );
          }
     }
}
```

备忘录角色的源代码如代码清单 12 所示，这个实现可以存储任意多的状态，外界可以使用检查点指数 index 来取出检查点上的状态。

代码清单 12：备忘录角色的源代码

```
package com.javapatterns.memento.whiteboxcheckpoints;
import java.util.Vector;
public class Memento
{
     private Vector states;
     private int index;

    /**
     *    构造子
     */
     public Memento(Vector states, int index)
     {
          this.states = (Vector) states.clone();
          this.index = index;
     }

    /**
     *    状态的取值方法
     */
     Vector getStates()
     {
          return states;
     }

    /**
     *    检查点指数的取值方法
     */
     int getIndex()
```

```
    {
        return this.index;
    }
}
```



备忘录的构造子克隆了传入的 states，然后将克隆存入到备忘录对象内部，这是一个重要的细节。因为不这样的话，将会造成客户端和备忘录对象持有对同一个 Vector 对象的引用，也可以同时修改这个 Vector 对象，会造成系统崩溃。

负责人的源代码如代码清单 13 所示。可以看出，负责人角色可以根据检查点指数 index 来恢复发起人角色的状态，也可以根据检查点指数 index 来取消一个检查点。

代码清单 13：负责人角色的源代码

```
package com.javapatterns.memento.whiteboxcheckpoints;
import java.util.Vector;
public class Caretaker
{
    private Originator o;
    private Vector mementos = new Vector();
    private int current;
   /**
    *    构造子
    */
    public Caretaker(Originator o)
    {
        this.o = o;
        current = 0;
    }
   /**
    *    创建一个新的检查点
    */
    public int createMemento()
    {
        Memento memento = o.createMemento();
        mementos.addElement(memento);
        return current++;
    }
   /**
    *    将发起人恢复到某个检查点
    */
    public void restoreMemento(int index)
    {
        Memento memento = (Memento)
            mementos.elementAt(index);
        o.restoreMemento(memento);
```

```
    }
   /**
    *     将某个检查点删除
    */
    public void removeMemento(int index)
    {
        mementos.removeElementAt(index);
    }
}
```

客户端角色的源代码如代码清单 14 所示。

代码清单 14：客户端角色的源代码

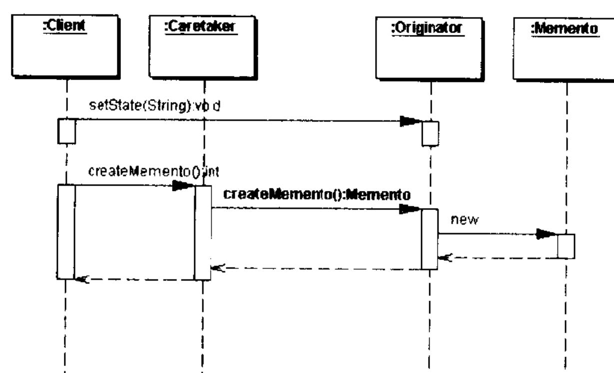
```
package com.javapatterns.memento.whiteboxcheckpoints;
public class Client
{
    private static Originator o = new Originator();
    private static Caretaker c = new Caretaker(o);
    static public void main(String[] args)
    {
        // 改变状态
        o.setState("state 0");
        // 建立一个检查点
        c.createMemento();
        // 改变状态
        o.setState("state 1");
        // 建立一个检查点
        c.createMemento();
        // 改变状态
        o.setState("state 2");
        // 建立一个检查点
        c.createMemento();
        // 改变状态
        o.setState("state 3");
        // 建立一个检查点
        c.createMemento();
        // 改变状态
        o.setState("state 4");
        // 建立一个检查点
        c.createMemento();
        // 打印出所有的检查点
        o.printStates();
        // 恢复到第二个检查点
        System.out.println("Restoring to 2");
        c.restoreMemento(2);
        o.printStates();
        // 恢复到第 0 个检查点
```

```
            System.out.println("Restoring to 0");
            c.restoreMemento(0);
            o.printStates();
            // 恢复到第三个检查点
            System.out.println("Restoring to 3");
            c.restoreMemento(3);
            o.printStates();
        }
    }
```

可以看出，客户端角色通过不断改变发起人角色的状态，并将之存储在备忘录里面，造成了五个检查点。通过指明检查点指数可以将发起人角色恢复到相应的检查点所对应的状态上。

时序图

将发起人的状态存储到备忘录对象中的活动序列图如下图所示。

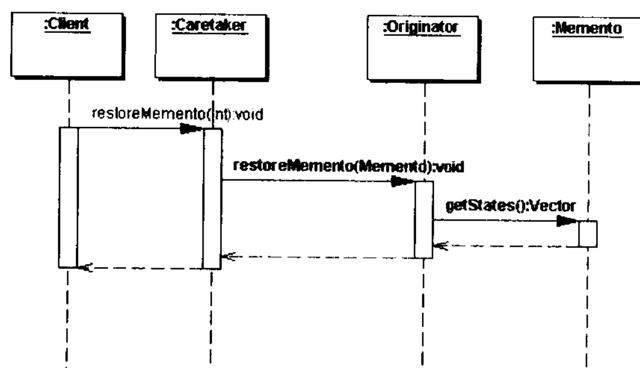


显然，将发起人对象的状态存储到备忘录中时，系统运行的时序是这样的：

（1）将发起人对象的状态设置成某个有效的状态；

（2）调用负责人角色的 createMemento()方法，负责人角色会负责调用发起人角色和备忘录角色，将发起人对象的状态存储起来。

将发起人对象恢复到某一个备忘录对象的检查点的活动序列图如下图所示。



由于负责人角色的功能被增强了，因此将发起人对象恢复到备忘录对象所记录的状态

时，系统运行的时序被简化了：

（1）将备忘录对象从负责人对象中取出。注意此时仅能得到 MementoIF 接口，因此无法读出或修改此对象的内部状态；

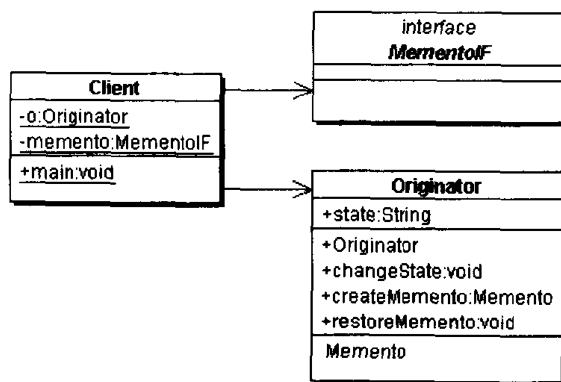
（2）将发起人对象的状态恢复成为备忘录对象所存储起来的状态，即“On”状态。由于发起人对象的内部类 Memento 实现了 MementoIF 接口，这个内部类是传入的备忘录对象的真实类型，因此发起人对象可以利用内部类 Memento 的私有接口读出此对象的内部状态。

49.7 “自述历史”模式

所谓“自述历史”模式（History-On-Self Pattern）实际上就是备忘录模式的一个变种[ANDERSON97]。在备忘录模式中，发起人（Originator）角色、负责人（Caretaker）角色和备忘录（Memento）角色都是独立的角色。虽然在实现上备忘录类可以成为发起人类的内部成员类，但是备忘录类仍然保持作为一个角色的独立意义。在“自述历史”模式里面，发起人角色自己兼任负责人角色。

设计类图和涉及的角色

“自述历史模式”的类图如下图所示。

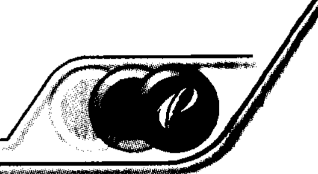


备忘录角色有如下的责任：

（1）将发起人（Originator）对象的内部状态存储起来。备忘录可以根据发起人对象的判断来决定存储多少发起人（Originator）对象的内部状态。

（2）备忘录可以保护其内容不被发起人（Originator）对象之外的任何对象所读取。备忘录有如下两个等效的接口。

- 宽接口：发起人对象可以看到一个宽接口（wide interface），这个宽接口允许它读取到备忘录的状态数据，以便根据这些数据恢复自己的内部状态。
- 窄接口：客户端对象看到的是备忘录的窄接口（narrow interface），这个窄接口只允许它把备忘录对象传给其他的对象，而不能读取备忘录的内部状态。



发起人角色有如下的责任：

（1）创建一个含有它当前的内部状态的备忘录对象。

（2）使用备忘录对象存储其内部状态。

客户端角色有负责保存备忘录对象的责任。

示意性的源代码

下面给出的是扮演窄接口角色的 MementoIF 接口的源代码，外部对象需要这个接口作为备忘录对象的类型和身份，如代码清单 15 所示。

代码清单 15：MementoIF 接口的源代码

```
package com.javapatterns.memento.historyonself;
public interface MementoIF
{
}
```

下面的发起人角色同时还兼任负责人角色，也就是说它自己负责保持自己的备忘录对象，如代码清单 16 所示。同时备忘录角色作为一个内部成员类出现在发起人类的内部，它的各个方法都是私有的，这样就保证了发起人角色自己可以存取备忘录的状态，而外部对象则不可以。

代码清单 16：Originator 类的源代码

```
package com.javapatterns.memento.historyonself;
public class Originator
{
    public String state;
   /**
    *    构造子
    */
    public Originator()
    {
    }
   /**
    *    改变状态
    */
    public void changeState(String state)
    {
        this.state = state;
        System.out.println("State has been changed to: "
            + state);
    }
   /**
    *    工厂方法，返还一个新的备忘录对象
    */
    public Memento createMemento()
```

```
    {
        return new Memento(this);
    }
   /**
    *    将发起人恢复到备忘录对象所记录的状态上
    */
    public void restoreMemento(MementoIF memento)
    {
        Memento m = (Memento) memento;
        changeState(m.state);
    }
   /**
    *    内部成员类，备忘录
    */
    class Memento implements MementoIF
    {
        private String state;
       /**
        *    状态的取值方法
        */
        private String getState()
        {
            return state;
        }
       /**
        *    构造子
        */
        private Memento(Originator o)
        {
            this.state = o.state;
        }
    }
}
```

下面是一个示意性的客户端角色的实现，如代码清单 17 所示。可以看出，由于没有独立的负责人角色，因此客户端的操作大大地简化了。

代码清单 17：Client 类的源代码

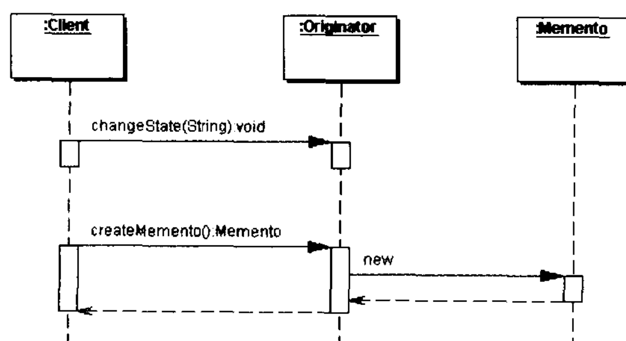
```
package com.javapatterns.memento.historyonself;
public class Client
{
    private static Originator o;
    private static MementoIF memento;
    public static void main(String[] args)
    {
        o = new Originator();
        // 修改状态
```



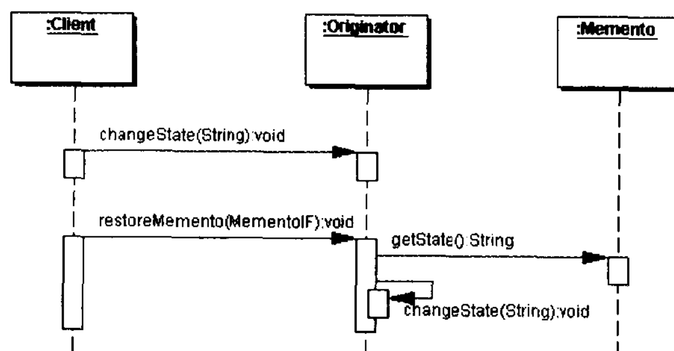
```
            o.changeState("State 1");
            // 创建备忘录
            memento = o.createMemento();
            // 修改状态
            o.changeState("State 2");
            // 按照备忘录恢复对象的状态
            o.restoreMemento(memento);
        }
    }
```

系统的时序图

将一个发起人的状态存储到一个新的备忘录对象中的活动序列图如下图所示。



将发起人对象恢复到一个存储在备忘录对象中的状态的活动序列图如下图所示。



与正常的备忘录模式的实现相比较，这里的角色少了一个，因此活动时序相应地变得简单了。

模式的优缺点

由于“自述历史”作为一个备忘录模式的特殊实现形式非常简单易懂，它可能是备忘录模式最为流行的实现形式。很多读者可能已经使用过这个形式，而没有意识到它是备忘录模式的一种。因此“简单易懂”是这一实现形式的最大优点。

对于通常有独立负责人角色的实现，如果系统中有多个发起人角色的话，这些发起人

角色可以共享同一个负责人角色，或者在多态性前提下使用多个负责人子类型。没有独立的负责人角色的"自述历史"实现无法做到这一点，因为每一个发起人都自己兼任负责人，无法做到代码共享，所以这种实现的缺点就体现出来了。

49.8 "假如"协议模式

文献[GRIFFIN93]讨论了一种称为"假如"协议的模式，这一模式实际上就是备忘录模式的另一个变种[ALPERT98]。在引进"假如"协议模式之前，首先考虑备忘录模式的操作过程。

备忘录模式的操作过程

备忘录模式的操作过程可以总结成以下的步骤：

（1）客户端为发起人角色创建一个备忘录对象。

（2）调用发起人对象的某个操作，这个操作是可以撤销的。

（3）检查发起人对象所处状态的有效性。检查的方式可以是发起人对象的内部自查，也可以由某个外部对象进行检查。

（4）如果需要的话，将发起人的操作撤销，也就是说根据备忘录对象的记录，将发起人对象的状态恢复过来。

"假如"协议模式的操作过程

文献[GRIFFIN93]描述了一个稍稍不同的操作步骤：

（1）将发起人对象做一个拷贝。

（2）在拷贝上执行某个操作。

（3）检查这个拷贝的状态是否有效和自恰。

（4）如果检查结果是无效或者不自恰的，那么扔掉这个拷贝，并触发异常处理程序；相反，如果检查结果是有效和自恰的，那么在原对象上执行这个操作。

显然，这后一种做法对于撤销一个操作并恢复操作前状态较为复杂和困难的发起人对象来说，是一个较为谨慎和有效的做法。

"假如"协议模式的优点和缺点

具体来说，这个做法的长处是可以保证发起人对象永远不会处于无效或不自恰的状态上。

这样做的短处是成功的操作必须执行两次。如果操作的成功率较低的话，这样做就比较划算，反之就不太划算。

在有些情况下，拷贝和原对象均有共享的数据，这时候无论修改拷贝还是修改原对象



都会导致数据被修改。此时“假如”协议模式就没有意义了。

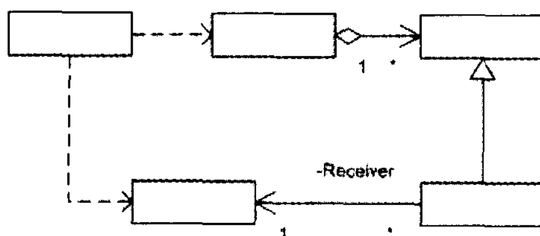
49.9　备忘录模式与其他模式的关系

备忘录模式与命令模式的关系

如果涉及到某个对象的可撤销操作的状态存储问题，那么仅仅使用备忘录模式是不够的，应当考虑使用命令模式。

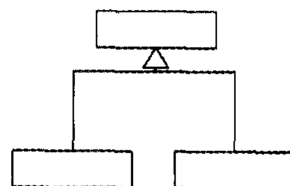
如果在某个系统中使用命令模式时，需要实现命令的撤销功能，那么命令模式可以使用备忘录模式来存储可撤销操作的状态。

命令模式的简略类图如下图所示。



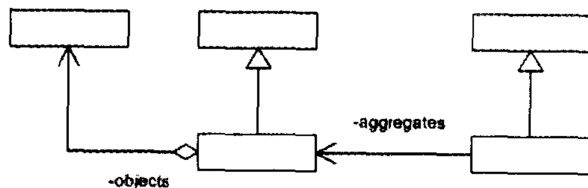
备忘录模式与原始模型模式的关系

如果发起人角色支持原始模型模式的话，备忘录模式可以使用原始模型模式进行备忘录的创建。原始模型模式的简略类图如右图所示。



备忘录模式与迭代子模式的关系

当备忘录模式支持多个检查点时，在各个检查点之间进行遍历需要使用迭代子模式。迭代子模式的简略类图如下图所示。



49.10 备忘录模式的应用

JDBC 与数据库

大多数的 Java 信息系统都是与数据库紧密相连的，这样的系统必然与 JDBC 库有密切的关系。其他的系统即便不使用数据库，也会使用某种存储方案，比如文件存储方案等。

无论是哪一种存储方案，在架构设计的时候都必须注意将商业逻辑与存储逻辑分割开来。这样做的原因如下：

（1）开发商业构件的技能与开发数据库构件的技能是完全不同的。将两者分割开来处理，意味着可以将它们交给不同的设计师，这显然对系统维护有好处。

（2）在商业逻辑变化的时候，存储方案不一定跟着变化。这样，系统的存储构件不必随着商业构件的变化而变化。

（3）在存储逻辑变化的时候，商业逻辑不一定跟着变化。比如一个系统可能因为存储量的关系必须更改 RDBMS 软件，比如从一个小型的 RDBMS 软件改换为一个中大型的 RDBMS 软件。将存储逻辑与商业逻辑分开，可以避免在更改存储方案时，影响商业逻辑构件。

J2EE 框架中备忘录模式的应用

J2EE 框架中的 Servlet 技术涉及到几个很有意义的模式，包括模版方法模式、观察者模式等，有兴趣的读者可以参阅本书的“专题：Servlet 技术中的模式”一章。Java Servlet 引擎提供 Session 对象，可以弥补 HTTP 协议的无状态（Stateless）缺点，存储用户的状态。本章探讨备忘录模式在 Servlet 技术中的应用。

什么是 Session 对象

由于 HTTP 协议是无状态的（stateless），而任何一个有意义的动态信息系统都需要在一定程度上保持用户的历史状态，也就是说必须是 stateful 的。因此 Web 技术发展出了三种不同的办法来解决状态的问题：Cookie，URL 改写（URL Re-writing）和 HTML 的隐藏表（Hidden Form）等办法。Session 对象是建立在这些技术的基础之上的，但是将实现的细节隐藏了起来。

当一个新的网络用户调用一张 JSP 网页或者 Servlet 时，Servlet 引擎会创建一个对应于这个用户的 Session 对象，具体的技术细节可能是向客户端浏览器发送一个含有 Session ID 的 Cookie，或者使用 URL 改写技术将 Session ID 包括在 URL 中等。

在一段有效的时间段里，同一个浏览器可以反复访问服务器，而 Servlet 引擎便可以使用这个 Session ID 来判断来访者应当与哪一个 Session 对象对应。



操作 Session 对象的 API

每一个 Session 对象都提供一套 put 和 get 方法，可以向 Session 对象存储信息，或者从中读取信息。Servlet API 提供了非常简便的方法来得到所对应的 Session 对象：

```
HttpSession currSession = request.getSession(flag);
```

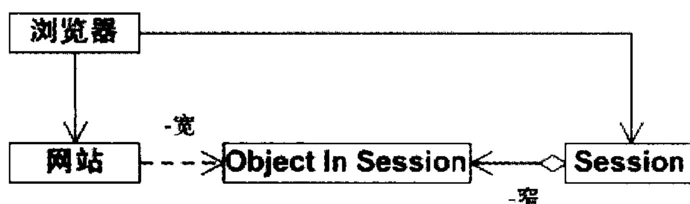
将一个对象加入到 Session 对象中去的办法也很简单，比如将 Name 对象加入到 currSession 对象中去：

```
currSession.putValue("userName", Name);
```

将一个事先存储好的对象从 Session 对象中取出的办法如下：

```
String storedName = currSession.getValue("userName");
```

显然 Session 对象就可以当做负责人角色使用，存入 Session 对象的任何信息都是备忘录角色，网站系统则是发起人角色，如下图所示。



用户输入数据

熟悉动态网站开发的读者一定非常熟悉下面所描述的做法。

每一个动态网站都会涉及到用户输入数据检查。这种检查不可能都在客户端利用 JavaScript 做到，有很多情况都是必须要服务器段检查的。

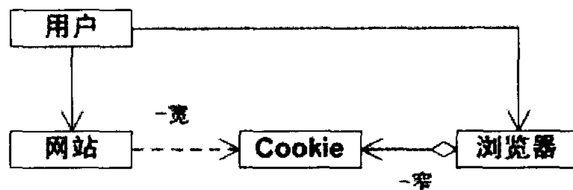
如果用户在填写一个表格时，输入了部分错误的数据，或者不完全的数据，那么网站系统应当记录下用户所输入的数据的正确部分，并自动填充到表格里面，呈现给用户，以便用户可以继续填写表格。

也就是说，在几次客户端与服务器端的通信过程中，系统需要记住用户输入的一些信息，并允许用户回访这些信息，并修改这些信息，而这就是备忘录模式的应用。

浏览器的 Cookie 文件

Cookie 是由网络服务器发送出来以存储在网络浏览器上的小量信息，从而下次用户有使用这个浏览器回到这个网络服务器时，网络服务器可从此浏览器读回此信息。Cookie 是很有用的，可以让浏览器记住这位访客的特定信息，如上次访问的位置、花费的时间或用户首选项等。

Cookie 是一个存储在浏览器目录中的文本文件，当浏览器运行时，存储在 RAM 中。一旦用户从该网站或网络服务器退出，Cookie 也可存储在计算机的硬驱上，Cookie 实际上就是备忘录角色，如下图所示。



显然，这也是备忘录模式的应用。浏览器扮演负责人角色，Cookie 便是备忘录角色，而网站便是发起人角色。网站可以通过浏览器读取 Cookie 中的信息，而浏览器仅仅保存传递来的 Cookie。

系统配置文件

一个软件系统有一些配置参数可能会存储在一个配置文件里面，包括一些用户偏爱的视窗位置、系统选项等。比如在视窗关闭时，软件系统会将视窗位置存储在配置文件中；在软件系统重新启动时，回收想从配置文件中读取的这些位置参数，并将视窗恢复到上一次关闭的状态上。这就是备忘录模式的应用。

49.11 使用备忘录模式的优点和缺点

备忘录模式的优点

使用备忘录模式有很多优点，下面是主要的三点：

- 有时一些发起人对象的内部信息必须保存在发起人对象以外的地方，但是必须要由发起人对象自己读取。这时，使用备忘录可以把复杂的发起人内部信息对其他的对象屏蔽起来，从而可以恰当地保持封装的边界。
- 本模式简化了发起人（Originator）类。发起人（Originator）不再需要管理和保存其内部状态的一个个版本，客户端可以自行管理它们所需要的这些状态的版本。
- 当发起人角色的状态改变的时候，有可能这个状态无效，这时候就可以使用暂时存储起来的备忘录将状态复原。

备忘录模式的缺点

备忘录模式也有几个缺点：

- 如果发起人角色的状态需要完整地存储到备忘录对象中，那么在资源消耗上面备忘录对象会很昂贵。
- 当负责人角色将一个备忘录存储起来的时候，负责人可能并不知道这个状态会占用多大的存储空间，从而无法提醒用户一个操作是否会很昂贵。



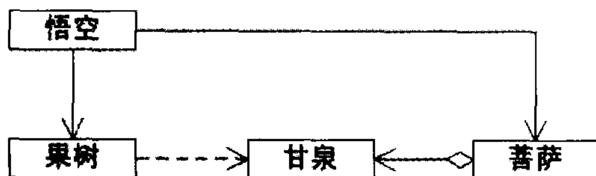
● 当发起人角色的状态改变的时候，有可能这个状态无效。如果状态改变的成功率不高的话，不如采取“假如”协议模式。

49.12　观世音甘泉活树的故事

却说孙大圣保唐僧西行，路过万寿山五庄观，与道童发生口角，一时发怒，把人参果树推倒。只见“那棵树倒在地下，土开根现，叶落枝枯。”

大圣只好请菩萨救活人参果树。菩萨将杨柳枝蘸出瓶中甘露，道：“扶起树来，从头浇下，自然根皮相合，叶长芽生，枝青果出。”菩萨将杨柳枝细细洒上，口中又念着经咒。不多时，洒净那舀出之水，只见那树果然依旧青枝绿叶浓郁阴森，上有二三十个人参果。

这就是说，果树的状态保存在菩萨的甘露之中，菩萨可以从甘露中将果树的状态恢复过来。这就是备忘录模式的应用，果树是发起人角色，甘泉是备忘录角色，而菩萨扮演负责人角色，如下图所示。



问答题

1. 我有两个网站，site1 和 site2。当一个用户进入 site1 时，我使用 Servlet 的 Session 对象跟踪它的 session。用户从 site1 换到 site2，过一段时间后又回到 site1。这时，用户的旧的 session 就被丢掉了。请问怎样才能保存老的 session 并在用户回到 site1 时仍然使用这个 session 呢？（本题及答案来自文献[DotCom]）

2. 请问带有 public、protected、（空）或者 private 等不同修饰符的方法之间在访问权限上有什么区别？

3. 请问带有 public、protected、（空）或者 private 等不同修饰符的内部成员类之间在访问权限上有什么区别？

4. 假设有一个普通（顶级）类 Outer，它有一个内部成员类，名为 Inner。Outer 和 Inner 均不是 Thread 类型的，并且 Inner 不是静态的。请给出一个示意性的源代码，并且回答下面的问题：当 Outer 的实例没有引用，但是 Inner 的实例仍然有引用时，Outer 的实例会不会被垃圾收集器收集走？

5. 本章的“备忘录的白箱实现”一节中给出了一个简单的单状态（获单检查点）备忘录模式的白箱实现。请参照“负责人角色的增强”一节所描述的，将这个实现中的负责人角色增强化。请给出设计类图、时序图以及源代码。

6. 本章前面所谈到的黑箱实现中的窄接口 MementoIF 一定不能有任何的方法吗？

7. 在黑箱和白箱实现中，备忘录对象是不是不变模式？

问答题答案

1. 这个问题可以有几种不同的解决方案。

第一种解决方案：将 session 的 timeout 值设置成-1，这样 session 将永不会失效。这样做的缺点是用户的浏览器关闭之后，session 也仍然存在，而新的浏览器不可能再与此 session 接上。因此这样做必须保证用户能自己将 session 关闭，如果用户只是将浏览器关闭而没有退出 site1 系统的话，此 session 将在侍服器上永远存在下去，直到侍服器重新启动为止。

第二种解决方案：当用户的 session 在 site1 失效时，可以使用备忘录模式将用户 session 存储起来。当用户再次访问 site1 系统时，可以将用户的状态重建起来。使用 HttpSessionBindingListener 接口接受 session 的事件，在 session 结束事件发生时，将 session 对象存储在文件或数据库里面。

这样，session 可以在该失效时失效，而不会有效率问题。

2. 当一个方法的修饰符是 public、protected、（默认）或者 private 时，代表这个方法的访问权限是 public、protected、package 或者 private 的，如下表所示。

可否访问	public	protected	package	private
所在类	可	可	可	可
同一 package 中的类	可	可	可	否
不同 package 中的子类	可	可	否	否
不同 package 中的非子类	可	否	否	否

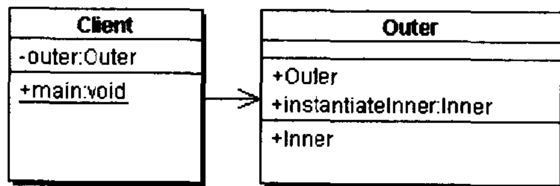
3. 带有 public、protected、（默认）或者 private 等不同修饰符的内部成员类之间在访问权限上的区别如下：

- privated 修饰符的内部成员类的访问权限等于 package 权限。
- protected 修饰符的内部成员类的访问权限等于 public 权限。

关于 public 权限和 package 权限请参见上题答案。

再重复一遍，内部成员类没有真正的 private 权限和 protected 权限。

4. 系统的类图如下图所示。



系统的源代码如代码清单 18 所示。

代码清单 18：Outer 类的源代码

```
package com.javapatterns.memento.innermember;
public class Outer
```

```
{
    /**
     *  构造子
     */
     public Outer()
     {
          System.out.println("Outer is instantiated.");
     }
    /**
     *  工厂方法，返还一个新的 Inner 对象
     */
     public Inner instantiateInner()
     {
          return new Inner();
     }
    /**
     *  内部成员类
     */
     public class Inner
     {
          public Inner()
          {
               System.out.println("Inner is instantiated.");
          }
          public void testThis()
          {
               System.out.println("This is this : "
                    + this);
               System.out.println("This is Outer.this : "
                    + Outer.this);
          }
     }
}
```

客户端类的源代码如代码清单 19 所示。

代码清单 19：客户端 Client 类的源代码

```
package com.javapatterns.memento.innermember;
public class Client
{
     private static Outer outer;
     public static void main(String[] args)
     {
          outer = new Outer();
          Outer.Inner inner = outer.instantiateInner();
          inner.testThis();
     }
```




```
}
```

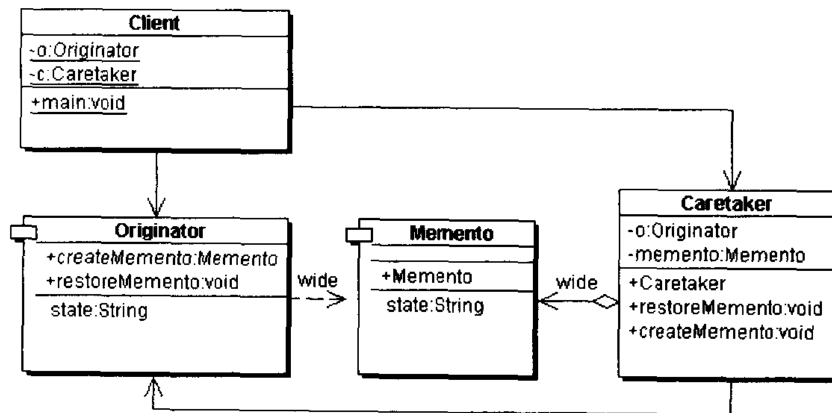
当 Inner 类不是静态的，而且 Outer 与 Inner 均不是 Thread 时，对问题的回答如下：

内部类 Inner 的每一个实例都必定是从某一个已经存在的 Outer 的实例中创建出来的。在 Inner 对象的内部会有一个指向生成它自己的 Outer 对象的指针。

读者想必都知道在 Inner 类内部关键字 this 方法所在的 Inner 对象，可能不是每一个读者都知道，在 Inner 类内部 Outer.this 是指向产生此 Inner 对象的 Outer 对象的。

当所有对一个 Outer 实例的引用都消失，仅剩下某个 Inner 的实例时，Outer 实例还有一个应用存放在那个 Inner 实例中，因此这个 Outer 的实例不会被收集。

5. 增强的负责人角色与系统的设计类图如下图所示。



可以看出，Caretaker 发生了如下的改动：首先取消了 retrieveMemento()和 saveMemento()两个方法，其次增加了 restoreMemento()和 createMemento()两个方法。由于 Caretaker 需要一个对 Originator 的引用，因此增加了一个构造子，用来接受一个 Originator 对象，如代码清单 20 所示。

代码清单 20：Originator 类的源代码

```
package com.javapatterns.memento.whiteboxenhanced;
public class Originator
{
    private String state;
   /**
    *  工厂方法，返还一个新的备忘录对象
    */
    public Memento createMemento()
    {
        return new Memento(state);
    }
   /**
    *  将发起人恢复到备忘录对象记录的状态上
    */
    public void restoreMemento(Memento memento)
    {
        this.state = memento.getState();
```



```
    }
  /**
   *  状态的取值方法
   */
   public String getState()
   {
        return this.state;
   }
  /**
   *  状态的赋值方法
   */
   public void setState(String state)
   {
        this.state = state;
        System.out.println("Current state = "
             + this.state);
   }
}
```

下面是 Memento 类的源代码，如代码清单 21 所示，可以看出这个类与负责人角色没有增强时相比没有变化。

代码清单 21：Memento 类的源代码

```
package com.javapatterns.memento.whiteboxenhanced;
public class Memento
{
    private String state;
  /**
   *  构造子
   */
   public Memento(String state)
   {
        this.state = state;
   }
  /**
   *  状态的取值方法
   */
   public String getState()
   {
        return this.state;
   }
  /**
   *  状态的赋值方法
   */
   public void setState(String state)
   {
        this.state = state;
```

```
        }
    }
```

下面的负责人角色是变化最大的，如代码清单 22 所示。首先 restoreMemento()方法取代了 retrieveMemento()方法，createMemento()方法取代了 saveMemento()方法；其次增加了一个接受 Originator 对象的构造子。

代码清单 22：Caretaker 类的源代码

```
package com.javapatterns.memento.whiteboxenhanced;
public class Caretaker
{
    private Originator o ;
    private Memento memento;
    /**
     *  构造子
     */
    public Caretaker(Originator o)
    {
        this.o = o;
    }
    /**
     *  将发起人恢复到备忘录纪录的状态上
     */
    public void restoreMemento()
    {
        this.o.restoreMemento(this.memento );
    }
    /**
     *  创建一个新的备忘录对象
     */
    public void createMemento()
    {
        this.memento = this.o.createMemento();
    }
}
```

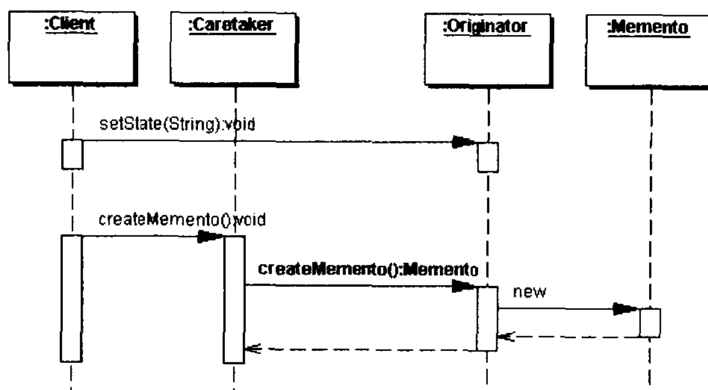
下面的客户端角色的变化值得注意。客户端操作负责人角色的能力增强了，通过操作负责人角色，客户端可以进行备忘录的创建和发起人状态的恢复，如代码清单 23 所示。

代码清单 23：Client 类的源代码

```
package com.javapatterns.memento.whiteboxenhanced;
public class Client
{
    private static Originator o = new Originator();
    private static Caretaker c = new Caretaker(o);
    public static void main(String[] args)
    {
```

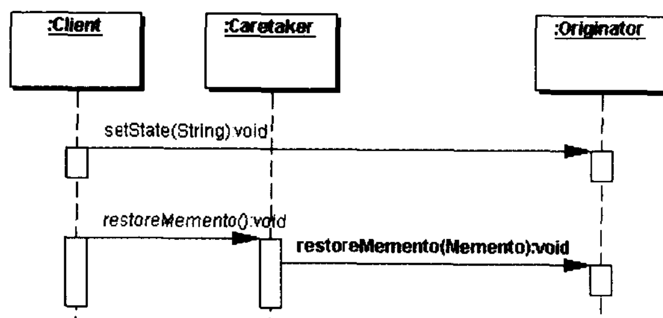
```
        // 改变负责人对象的状态
        o.setState("On");
        // 创建备忘录对象，并将发起人对象的状态存储起来
        c.createMemento();
        // 修改发起人对象的状态
        o.setState("Off");
        // 恢复发起人对象的状态
        c.restoreMemento();
    }
}
```

当一个新的状态被存储起来时，系统的时序图如下图所示。



可以看出，客户端通过直接控制负责人对象就可以创建一个备忘录，上面的时序图与负责人没有增强的时候相比更加流畅。

当发起人被恢复到某一个被存储的状态时，系统的时序图如下图所示。



显然，客户端通过直接控制负责人对象就可以将发起人恢复到一个备忘录对象所记录的状态上，上面的时序图与负责人没有增强的时候相比更加流畅。

6. 备忘录模式对黑箱实现中的 MementoIF 接口的要求是它不可以破坏封装。所谓封装，是指对备忘录对象所存储的发起人状态的封装。

一般而言，封装是在两个层次上：第一个层次是不允许修改，但允许读取；第二个层次是既不允许修改，也不允许读取。两个层次对 MementoIF 接口的要求也是不同的，第一个层次的封装要求 MementoIF 接口不能提供任何修改备忘录对象所存储的发起人状态的方法，但是可以提供阅读发起人状态的方法；第二个层次的封装要求 MementoIF 接口

不提供任何阅读和修改的方法。设计师需要根据所设计的系统的需要，决定选择哪一个层次的封装。

两个层次的封装都没有说 MementoIF 接口不能声明任何方法。只要 MementoIF 接口声明的方法并不违反所选定的封装层次就可以了。

当然，一个 Java 接口本身并不能说明一个方法是否会破坏封装。要看清楚这一点需要考察这个方法的具体实现，也就是实现 MementoIF 接口的具体 Memento 类的源代码。

7. 不是。

参考文献

[DotCom] Dot-Com Builder. Technology Guides – Java. HTTP://dcb.sun.com/technology/java/qa_archive/

[GRIFFIN93] Susan Griffin. What if? A protocol for object validation. Smalltalk Report, 3:3, November 1993

[ANDERSON97] Francis Anderson and Ralph Johnson. Tree with History. paper submitted to PloP, 1997

第 50 章　状态（State）模式

状态模式（State Pattern），又称状态对象模式（Pattern of Objects for States）。状态模式是对象的行为模式[GOF95]。

状态模式允许一个对象在其内部状态改变的时候改变其行为。这个对象看上去就像是改变了它的类一样。

50.1　引　言

1979 年在湖北出土的曾侯乙编钟，一套共 65 件，总音域跨 5 个八度，12 个半音齐全，每一只钟都发出不同的音。曾侯乙编钟的复制件如下图所示。



用面向对象的语言来讲，编钟有能够动态变化的属性，也就是它所发出的声音。编钟的这种属性叫做状态，而编钟也因为有这样的属性而被叫做有状态的（stateful）对象。编钟所发出的声音是由敲击的那一只钟决定的，而钟的数目和每一只钟的状态是事先确定的。换言之，这些钟代表了编钟的所有可能状态。

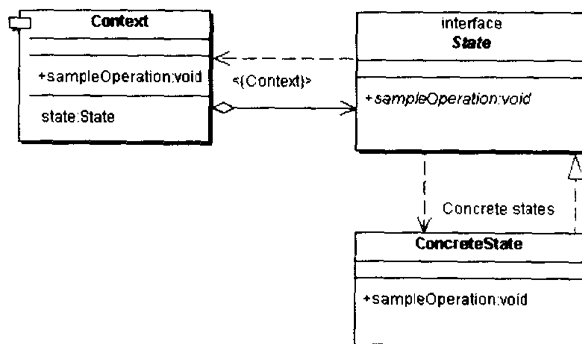
有很多情况下，一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做状态，这样的对象叫做有状态的（stateful）对象。这样的对象状态是从事先定好的一系列值中取出的。当一个这样的对象与外部事件产生互动时，其内部状态就会改变，从而使得系统形为也随之发生变化。

当乐师不断打击一只只钟时，编钟的状态由一支钟过渡到另一只钟。编钟的行为，也就是钟所发出的声音就会从一支钟过渡到另一支钟。编钟就是这样在乐师们的打击下奏出音乐的。

50.2　状态模式的结构

用一句话来表述，状态模式把所研究的对象的行为包装在不同的状态对象里，每一个

状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式的示意性类图如下图所示。



模式所涉及的角色有：

- 抽象状态（State）角色：定义一个接口，用以封装环境（Context）对象的一个特定的状态所对应的行为。
- 具体状态（ConcreteState）角色：每一个具体状态类都实现了环境（Context）的一个状态所对应的行为。
- 环境（Context）角色：定义客户端所感兴趣的接口，并且保留一个具体状态类的实例。这个具体状态类的实例给出此环境对象的现有状态。

在上图可以看出，环境类 Context 是所考察的类，而它的行为 sampleOperation()是委派给类型为 State 的一个对象的。由于 State 本身是一个抽象类或接口，实际担当此任务的是具体状态类，即 ConcreteState。上图中只给出了一个具体状态类，而实际情况中会有很多具体状态类的。换言之，环境类 Context 的行为 sampleOperation()是委派给某一个具体状态类的。

通过使用多态性原则，可以动态地改变环境类 Context 的属性 State 的内容，使其从指向一个具体状态类变换到指向另一个具体状态类，从而使环境类的行为 sampleOperation()由不同的具体状态类来执行。

本模式的示意性源代码如代码清单 1 所示。

代码清单 1：环境类 Context 的源代码

```
public class Context
{
    private State state;
    public void sampleOperation()
    {
        state.sampleOperation();
    }
    public void setState(State state)
    {
        this.state = state;
    }
}
```

上面就是环境角色的源代码，如代码清单 2 所示。可以看出，环境类持有一个 State 对象，并把所有的行为委派给此对象。

代码清单 2：抽象状态 State 接口的源代码

```
public interface State
{
    void sampleOperation();
}
```

这个接口规范所有实现此接口的子类，要求它们都实现方法 sampleOperation()，如代码清单 3 所示。

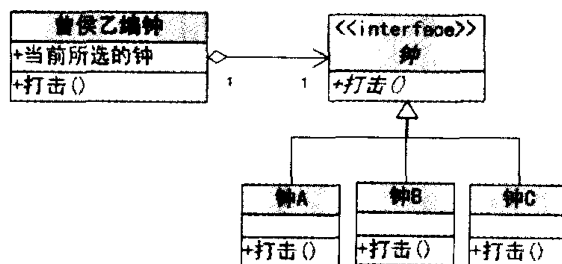
代码清单 3：具体状态类 ConcreteState 的源代码

```
public class ConcreteState implements State
{
    public void sampleOperation(){ }
}
```

可以看到，这个类真正实现了 sampleOperation()方法。

50.3 曾侯乙编钟的描述

曾侯乙编钟系统可以用 UML 来表述，如下图所示。



曾侯乙编钟系统所涉及的角色有：

- 抽象状态（State）角色：此角色由“钟”接口扮演。这个接口用以规范所有的钟的行为。
- 具体状态（ConcreteState）角色：此角色由一个个的“钟 A”，“钟 B”，“钟 C”扮演。每一个钟都有特定的发声频率。
- 环境（Context）角色：定义出“曾侯乙编钟”，并且保有一个钟的实例（即“当前所选的钟”属性）。这个实例给出此编钟现在发的声。

本例的示意性源代码如代码清单 4 所示。

代码清单 4：环境类“曾侯乙编钟”的源代码

```
public class 曾侯乙编钟
{
```

```
        private 钟 state;
        public void 打击()
        {
            state.打击();
        }
        public void setState(钟 state)
        {
            this.state = state;
        }
    }
```

乐师用眼睛选中一只钟，然后打击它，如代码清单 5 所示。

代码清单 5："钟"接口的源代码

```
public interface 钟
{
    void 打击();
}
```

显然，此接口要求所有的状态子类都定义行为"打击()"，如代码清单 6 所示。

代码清单 6：具体状态类"钟 A"的源代码

```
public class 钟 A implements 钟
{
    public void 打击()
    {
        //钟受打击而发声的源代码
    }
}
```

"打击()"行为在这里被实现。这个类就如同 Netscape 的 Steve Abell 所说，没有数据，只有行为。

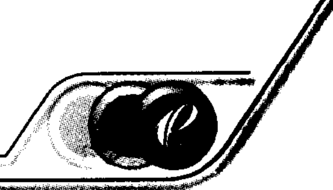
"曾侯乙编钟"的"打击()"行为依赖于状态 state。而 state 的数据类型是抽象状态类，即"钟"接口。任何具有此接口的对象都可以为"曾侯乙编钟"提供"打击()"行为。显然，"钟 A"、"钟 B"、"钟 C"是满足条件的。

由于使用了状态模式，曾侯乙编钟的设计者把编钟的行为与状态的依赖巧妙地通过委派，把编钟的行为委派给一只只的钟。这就是使用面向对象的语言对曾侯乙编钟的解释。

50.4 状态模式的效果

状态模式的效果如下所示。

（1）状态模式需要对每一个系统可能取得的状态创立一个状态类（State）的子类。当系统的状态变化时，系统便改变所选的子类。所有与一个特定的状态有关的行为都被包装到一个特定的对象里面，使得行为的定义局域化。因为同样的原因，如果有新的状态以



及它对应的行为需要定义时，可以很方便地通过设立新的子类的方式加到系统里，不需要改动其他的类。

（2）由于每一个状态都被包装到了类里面，就可以不必采用过程性的（procedural）处理方式，使用长篇累牍的条件转移语句。

（3）使用状态模式使系统状态的变化变得很明显。由于不用一些属性（内部变量）来指明系统所处的状态，因此，就不用担心修改这些属性不当而造成的错误。

（4）可以在系统的不同部分使用相同的一些状态类的对象。这种共享对象的办法是与享元模式相符合的。事实上，此时这些状态对象基本上是只有行为而没有内部状态的享元模式。

（5）状态模式的缺点是会造成大量的小的状态类；优点是使程序免于大量的条件转移语句，使程序实际上更易于维护。

（6）系统所选的状态子类均是从一个抽象状态类或接口继承而来，Java 语言的特性使得在 Java 语言中使用状态模式较为安全。多态性原则是状态模式的核心。

50.5　在什么情况下使用状态模式

在以下各种情况下可以使用状态模式：

（1）一个对象的行为依赖于它所处的状态，对象的行为必须随着其状态的改变而改变。

（2）对象在某个方法里依赖于一重或多重的条件转移语句，其中有大量的代码。状态模式把条件转移语句的每一个分支都包装到一个单独的类里。这使得这些条件转移分支能够以类的方式独立存在和演化。维护这些独立的类也就不再影响到系统的其他部分。

50.6　关于模式实现的讨论

本模式在实现时有以下这些值得注意的地方：

（1）谁来定义状态的变化。状态模式并没有规定哪一个角色决定状态发生转换的条件。如果转换条件是固定的，那么决定就应当由 Context 角色来做。以编钟为例，如果编钟要演奏的曲子是固定的，那么就可以把曲子存储在编钟里，由编钟自己演奏，决定所发的音的转换。

然而，如果让 State 子类自行决定它的下一个继任者是谁，以及在什么时候进行转换，就更有灵活性。仍然以编钟为例，可以把程序存储在每一个钟里，由每一个钟自行决定下面该发生的钟是哪一个，以及该在什么时候发声，这样就更有灵活性。

或者，由外界事件来决定状态的转换。也就是说，把编钟的演奏交给系统外部的演奏师。

（2）状态对象的创立和湮灭。其中值得权衡的做法有两种：

- 动态地创立所需要的状态对象，不要创立不需要的状态对象。当不再需要某一状

态对象时，便马上把此状态对象的实例湮灭掉。这相当于在需要某一只钟时，才把这只钟挂在编钟上；当这只钟使用完后，就立即把它卸下来；当再次使用时，再把它挂回去。这对编钟来说不是一个好主意，因为把那些很沉的编钟搬上搬下实在不方便。但是对于其他的一些系统，在事先不知道要使用哪一个状态对象，而一旦开始使用便不再频繁变更的情况下，这显然是一个好主意。

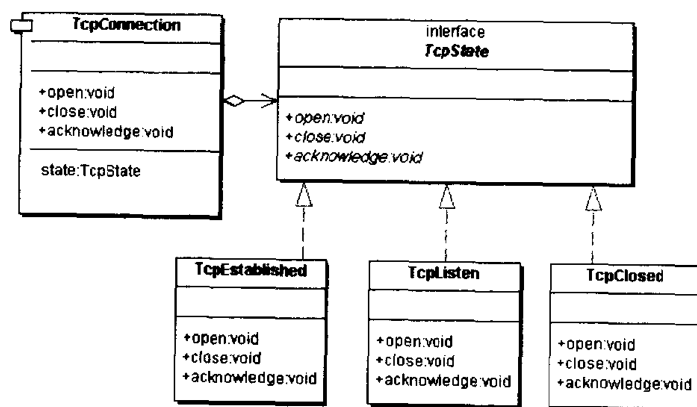
- 事先创立所有的对象，然后不再湮灭它们。这相当于把事先说的钟都挂在编钟上，然后在演奏过程当中并不把它们卸下来。这显然适用于状态变化比较快和频繁、加载这些状态对象的成本较高的情况，比如编钟系统。

对于 Java 系统，湮灭一个对象不是由程序完全控制的。当一个对象不再被引用时，Java 语言的垃圾处理器会自动把它湮灭掉。但是具体在什么时候湮灭，则不是可以控制的，甚至不是可以预料的。

（3）环境类可以把它自己作为参量传给状态对象。这样，一旦需要，状态对象就可以调用环境对象。

50.7 一个状态模式的应用例子：TCP

这是一个状态模式的经典例子。考虑由 TcpConnection 代表一个 TCP/IP 网络的连接。一个 TcpConnection 对象可以取几个可能的状态之一：Establised（已建立连接）、Listening（聆听）和 Closed（关闭）。当 TcpConnection 对象接到其他的对象的请求时，会根据其状态不同而给出不同的回应。比如说，TcpConnection 回应客户端的开启请求取决于 TcpConnection 的状态是 Establised 还是 Closed。状态模式描述了 TcpConnection 在它的状态下是怎样行动的。TCP 系统的 UML 类图如下图所示。



在此系统中，使用状态模式的关键是引进了一个抽象类或接口 TcpState 来代表网络的连接状态。抽象类或接口 TcpState 限定了所有的表示不同行为的状态，它的子类实现了由状态决定的行为。比如，TcpEstablished 和 TcpClosed 类分别实现了对应于 Established 和 Closed 状态的行为。

TcpConnection 类持有一个状态对象，亦即 TcpState 的一个子类的实例，来表征 TCP 连接的现在状态。TcpConnection 类把所有与状态有关的请求都委派给它的状态对象。

TcpConnection 使用它的 TcpState 的子类实例来执行特定的连接状态所对应的操作。

当连接的状态改变时，TcpConnection 对象就改变它所用的状态对象。例如，当网络连接从“已建立”改为“关闭”时，TcpConnection 对象会把它的状态对象从 TcpEstablished 的实例改为 TcpClosed 的实例。

在上图可以看出，系统有如下的角色：

- 环境角色：由 TcpConnection 扮演。此类定义了客户端感兴趣的接口，并持有一个具体状态类的实例，以定义它当前所处的状态。
- 状态角色：由 TcpState 扮演。此类定义了把依赖于一个特定的状态所对应的行为包装起来所需要的接口。
- 具体状态角色：由 TcpEstablished、TcpListen、TcpClosed 等扮演。它们中的每一个都实现了环境类的状态所对应的行为。

本例的示意性源代码如代码清单 7 所示。

代码清单 7：环境类 TcpConnection 的源代码

```
public class TcpConnection
{
    private TcpState state;
    public void open()
    {
        state.open();
    }
    public void setState(TcpState state)
    {
        this.state = state;
    }
    public void close()
    {
        state.close();
    }
    public void acknowledge()
    {
        state.acknowledge();
    }
}
```

显然，TcpConnection 类扮演了环境角色。下面的 Java 接口 TcpState 扮演了抽象的状态角色，如代码清单 8 所示。

代码清单 8：抽象状态类 TcpState 的源代码

```
public interface TcpState
{
    void open();
    void close();
    void acknowledge();
```

```
}
```

而下面的 TcpEstablished 是一个具体状态类，它实现了抽象状态角色所规定出的接口，如代码清单 9 所示。

代码清单 9：具体状态类 TcpEstablished 的源代码

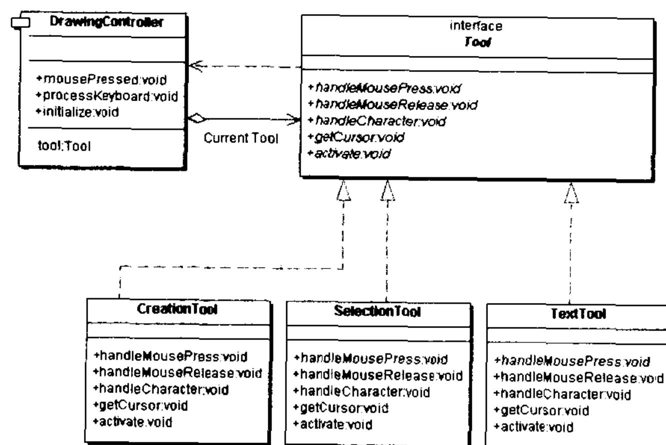
```
public class TcpEstablished
    implements TcpState
{
    public void open()
    {
        //write your code here
    }
    public void close()
    {
        //write your code here
    }
    public void acknowledge()
    {
        //write your code here
    }
}
```

TcpListen 和 TcpClose 类与此类均很相像，故省略。

最后，应当指出的是，本例只是 TCP 协议的一个示例性的实现。最早提出这个示意性实现的文献是[JZ91]，并为[GoF]的书所引述。

50.8 状态模式在绘图软件中的应用

在一些绘图软件系统里，状态模式被用做编辑器框架。如下图所示便是一个示意性的静态结构图。



这样的设计使得客户端能够轻易地挂接新的工具。只要准备好一个新的工具子类，那么就可以直接从 DrawingController 类引用这一新的工具类。

50.9　用户登录子系统

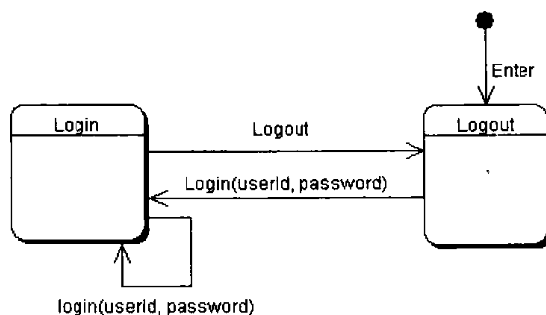
这里依据状态模式设计一个根据客户端传来的用户名和密码登录用户的子系统。用户记入系统时首先遇到登录网页，用户需要输入用户名和密码，单击【Log On】键，系统会向数据库查询。只要用户名和密码是正确无误的，用户就登录成功了，如下图所示。



当用户登录成功后，就会见到如下图所示的欢迎网页。相反，如果登录失败时，用户仍然要面对登录网页。



如下图所示，此状态图描述了子系统的状态变化情况。

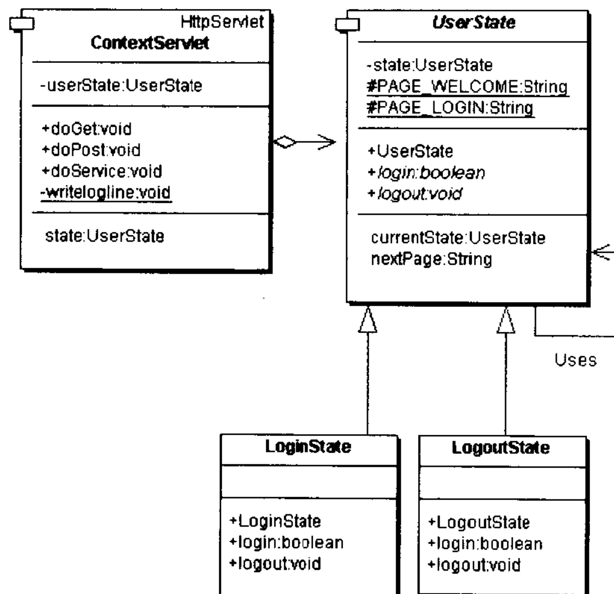


从上面的状态图可以看出，用户进入系统时，处于 Logout 状态，用户使用正确的用户名和密码便可以过渡到 Login 状态。当用户处于 Login 状态时，可以发出 Logout 命令，使系统过渡到 Logout 状态。用户在 Login 状态时，重新发出 Login 的命令不会改变状态。

不同的状态对象会产生不同的网页，对应于 Login 状态的状态对象给出欢迎网页，因为客户已经成功地登录系统；对应于 Logout 的状态对象重新给出登录网页，以便用户重

新登录。子系统需要一个登录网页作为客户端，登录网页接收用户名和密码，并在用户按下一个命令键时提交给一个 Servlet。这个 Servlet 便可以在 doPost()或 doGet()方法中接收用户名和密码，并且创立对应的状态对象。

根据上面的要求，设计出登录子系统的静态结构图，如下图所示。



下面就从客户端开始，仔细考察一下系统是怎样实现的，如代码清单 10 所示。登录网页在 submit 时的目标网页是环境类，即 Servlet（com.javapatterns.state.login.Context Servlet）。其中 submit 键的名字是“Log On”。

代码清单 10：ServletContextServlet 类的源代码

```
<HTML>
<BODY>
<FORM action="/Servlet/com.javapatterns.state.login.ContextServlet"
    method=get>
    User ID <INPUT TYPE=text name=userid value="jeff"><br>
    Password <INPUT TYPE=password name=password value="pass">
    <INPUT TYPE=submit name="btnAction" value="Log On">
</form>
</BODY>
```

与登录网页一样，欢迎网页在 submit 时的目标网页是环境类，即 Servlet（com.javapatterns.state.login.ContextServlet），如代码清单 11 所示。但是，其中 submit 键的名字是“Log Out”，这个区别很重要。Servlet 里利用这个区别判断客户端的请求使登录请求还是结束请求。

代码清单 11：ServletContextServlet 类的源代码

```
<HTML>
<BODY>
<form action="/Servlet/com.javapatterns.state.login.ContextServlet"
    method=get>
```

```
    Welcome to website! <br>
    <input type=submit name="btnAction" value="Log Out">
</form>
</BODY>
</HTML>
```

环境类的角色由 ContextServlet 类扮演，如代码清单 12 所示。环境类持有一个状态类的引用。这个引用最初指向一个 LogoutState 类的实例，这是因为最初的状态是 logout 状态。在用户的登录成功后，此引用会变成一个 LoginState 的实例。

代码清单 12：ContextServlet 类的源代码

```
package com.javapatterns.state.login;
import java.io.IOException;
import javax.Servlet.ServletException;
import javax.Servlet.ServletRequest;
import javax.Servlet.http.*;
public class ContextServlet extends HttpServlet
{
    // 持有一个对状态的引用
    private UserState userState = new LogoutState();

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        doService(request, response);
    }
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        doService(request, response);
    }
    public void doService(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        String userId = request.getParameter("userid");
        String password = request.getParameter("password");
        String btnAction = request.getParameter("btnAction");
        if ( btnAction.equalsIgnoreCase("log on") )
        {
            this.getState().login(userId, password);
        }
        else if (btnAction.equalsIgnoreCase("log out"))
        {
```

```
            this.getState().logout();
        }
        response.sendRedirect(this.getState().getNextPage());
    }
    /**
     * 状态的取值方法
     */
    public UserState getState()
    {
        return userState.getCurrentState();
    }
}
```

环境类首先判断客户端的请求是登录还是退出。当接到登录请求时，环境类便调用状态类的 login()方法。相反，当接到退出请求时，环境类便调用状态类的 logout()方法。抽象类 UserState 扮演的就是抽象的状态类角色，如代码清单 13 所示。

代码清单 13：抽象类 UserState 的源代码

```
public abstract class UserState
{
    private UserState state;
    private String nextPage;
    protected static String PAGE_WELCOME =
        "/javapatterns/state/welcome.html";
    protected static String PAGE_LOGIN =
        "/javapatterns/state/login.html";
    /**
     * 构造子
     */
    public UserState()
    {
        this.nextPage = PAGE_LOGIN;
    }
    /**
     * 行为方法
     */
    public abstract boolean login(
        String userId, String password);
    /**
     * 行为方法
     */
    public abstract void logout();
    /**
     * 状态的赋值方法
     */
    public void setCurrentState(UserState state)
```

```
    {
        this.state = state;
    }
    /**
     * 状态的取值方法
     */
    public UserState getCurrentState()
    {
        if (this.state == null)
        {
            this.state = new LoginState();
        }
        return this.state;
    }
    public String getNextPage()
    {
        return nextPage;
    }
    public void setNextPage(String nextPage)
    {
        this.nextPage = nextPage;
    }
}
```

子系统里有两个具体状态类，即 LoginState 和 LogoutState，它们分别包装了系统在登录成功后和登录失败后的 Login 和 Logout 状态。当用户尚没有登录时，状态也是 Logout 状态。LoginState 类的源代码如代码清单 14 所示。

代码清单 14：状态类 LoginState 的源代码

```
public class LoginState extends UserState
{
    /**
     * 构造子
     */
    public LoginState()
    {
        //write code here
    }
    /**
     * 行为方法
     */
    public boolean login(String userId, String password)
    {
        setNextPage(UserState.PAGE_WELCOME);
        setCurrentState( new LoginState() );
        return true;
```

```
    }
    /**
     * 行为方法
     */
    public void logout()
    {
        setNextPage(UserState.PAGE_LOGIN);
        setCurrentState( new LogoutState() );
    }
}
```

正如状态模式所要做的，LoginState 包装了登录成功状态下子系统的行为，也就是再登录和退出行为。LogoutState 的源代码如代码清单 15 所示。

代码清单 15：状态类 LogoutState 的源代码

```
public class LogoutState extends UserState
{
    /**
     * 构造子
     */
    public LogoutState()
    {
        //write your code here
    }
    /**
     * 行为方法
     */
    public boolean login(String userId, String password)
    {
        StringBuffer sql = new StringBuffer(50);
        sql.append("SELECT COUNT(*) FROM user_info WHERE user_id = '")
            .append(userId)
            .append("' AND password = '")
            .append(password)
            .append("'");
        int counting = DBManager.getAggregate(sql.toString());
        if(counting > 0)
        {
            this.setNextPage(UserState.PAGE_WELCOME);
            this.setCurrentState(new LoginState());
            return true;
        }
        else
        {
            this.setNextPage(UserState.PAGE_LOGIN);
            this.setCurrentState(new LogoutState());
            return false;
```

```
        }
    }
    /**
     * 行为方法
     */
    public void logout()
    {
        this.setCurrentState(new LogoutState());
        this.setNextPage(UserState.PAGE_LOGIN);
    }
}
```

LogoutState 包装了未成功登录的状态下子系统的行为，包括登录和退出行为。本类的登录操作是系统真正检查用户名和密码正确性的地方。

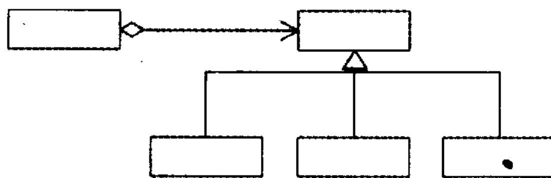
可以看出，状态的转移、何时进行转移和向哪一个状态进行转移，是由具体状态类 LogoutState 和 LoginState 决定的。

读者可能会注意到，上面的源代码中引用了一个 DBManager 类的方法 getAggregate (String sql)。DBManager 是一个工具类，负责和数据库打交道。setAggregate(String sql)方法是一个静态的方法，它接受一个字符串参数，也就是 SQL 语句，然后向数据库查询。此方法只返还记录集合的第一行的第一列。那么记录集合会不会根本就是空的呢？不会，因为做统计用的 SQL 语句必定会返还至少一行数据，而且往往只有一行和一列，这便是笔者设计这个方法的目的。

由于与本文的内容不是很有关联，因此，本章不打算涉及 DBManager 类以及数据库的结构设计，就假设它们都存在吧。

50.10　状态模式与策略模式的区别

状态模式经常与策略模式相混淆。有的时候很难区分应当使用状态模式还是应当使用策略模式。策略模式的简略类图如下图所示。



这个时候，一个简单的方法便是考察环境角色是否有明显的状态和状态的过渡。如果环境角色只有一个状态，那么就应当使用策略模式。策略模式的特点是：一旦环境角色选择了一个具体策略类，那么在整个环境类的生命周期里它都不会改变这个具体策略类。而状态模式则适用于另一种情况，即环境角色有明显的状态转移。在环境类的生命周期里面，会有几个不同的状态对象被使用。

另一个微妙的区别在于，策略模式的环境类自己选择一个具体策略类；而状态模式的环境类是被外在原因放进一个具体状态中。

策略模式所选的策略往往并不明显地告诉客户端它所选择的具体策略；而状态模式则相反，在状态模式里面，环境角色所处的状态是明显告诉给客户端的。

问答题

1. 请给出 DBManager 和 getAggregate()的一个示范性实现。
2. 在代码清单 15 中使用了 StringBuffer 来构造 SQL 语句。请问为什么不直接使用 String 来构造呢？
3. 请使用状态图描述一个门的状态。

问答题答案

1. 这里给出一个示范性的实现，如代码清单 16 所示。

代码清单 16：DBManager 的示意性源代码

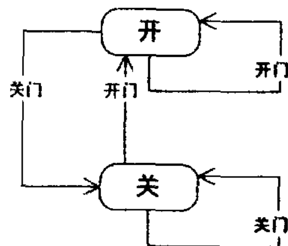
```
public class DBManager
{
    public static int getAggregate(String sql)
    {
        return 1;
    }
}
```

2. String 对象是不变对象。换言之，一个 String 实例的内容不能改变，如果内容一定要改变的话，就必须创造一个新的 String 实例。这意味着在用几个 String 构造出 SQL 语句时，会有效率问题。

关于不变模式请见“不变（Immutable）模式”一章。

3. 一个门有两个状态：开和关。这两个状态分别由两个矩形代表。

门的这两个状态在两个事件下发生过渡：开门事件和关门事件。开门事件使得开状态过渡到自己，并使得关状态过渡到开状态；关门事件使得关状态过渡到自己，并使得开状态过渡到关状态。其状态图如右图所示。



参考文献

[JZ91]R. E. Johnson and J. Zweig. Delegation in C++. Journal of Object-Oriented Programming，4(11):22-35，November 1991

第 51 章　专题：崂山道士与状态模式

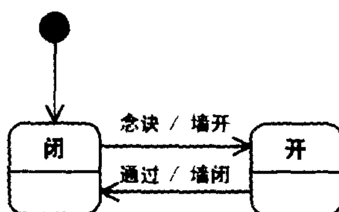
本章拟以模式设计的原理探索《聊斋志异》的情趣。

51.1　有状态的墙

大家都对墙在现实生活中的行为很熟悉，这里是指当人的脑袋碰上去时，墙的行为。这个行为是不会改变的，这就是说，墙是没有状态的，它属于无状态（stateless）对象。但是在《聊斋志异》中墙是有状态的（stateful）。“崂山道士”一节里讲到：

“道士……乃传以诀……呼曰：“入之！”……王果去墙数步，奔而入；及墙，虚若无物；回视，果在墙外矣。”

用面向对象的语言来讲，墙是有状态的（stateful）对象，墙的状态有开和闭两种。正确的口诀是墙的状态发生转变的外界事件，而墙在王生的头碰上来时的行为是由墙的状态决定的。换言之，对于崂山道士而言，墙的状态可以由下图所示的状态图来描述。



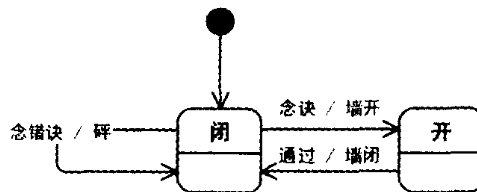
状态过渡连线的标签分为两部分，由斜线隔开。第一部分是激发状态过渡的外界事件，而第二部分则是对象在事件发生时所采取的行为。只要口诀是正确的，那么上面的状态图就是正常的逻辑。

51.2　异 常 逻 辑

所谓异常逻辑，是指出现异常时，系统的行为逻辑。如果口诀念错了，或是根本没有念诀，墙在头碰上来的时候的行为会怎么样呢？蒲松龄是这样描述的：

“去墙数尺，奔而入，头触硬壁，蓦然而踣。妻扶视之，额上坟起，如巨卵焉。”

听上去是挺惨的。使用面向对象的语言来讲，“念错诀”是引起状态过渡的外界事件。此事件引起对象（也就是墙）的行为是“砰”地让王生额上坟起巨卵。而墙仍然是“闭”状态的。过渡的方向仍然是“闭”状态本身，也就是说过渡关系是反身关系念错口诀时墙的状态图如下图所示。



51.3 从代码重构的角度考察

首先，实现这个系统最显然的方法就是使用多重的条件转移语句，如代码清单 1 所示。

代码清单 1：念错口诀时墙的状态源代码

```
static final int STATE_LOCKED = 1;
static final int STATE_UNLOCKED = 2;
static final int EVENT_SPELL = 4;
static final int EVENT_PASS = 8;
public void Transition(int event)
{
    int state = STATE_LOCKED;
    switch(state)
    {
    case (STATE_LOCKED):
        switch(event)
        {
        case (EVENT_SPELL):
            state = STATE_UNLOCKED;
            break;
        case (EVENT_PASS):
            Bang();
            break;
        }
        break;
    case STATE_UNLOCKED:
        switch(event)
        {
        case (EVENT_SPELL):
            state = STATE_UNLOCKED;
            break;
        case (EVENT_PASS):
            state = STATE_LOCKED;
            break;
        }
        break;
    }
```

```
	}
	public void Bang()
	{
		System.out.println("You are damned.");
	}
```

在上面的代码中，使用了两个常数代表两个状态，即“开”state_locked 和“闭”state_unlocked；另外两个常数代表两个事件，即“念诀”event_spell 和“跃入”event_pass。

如果老道士的法术变得更加复杂时，怎么处理呢？比如：

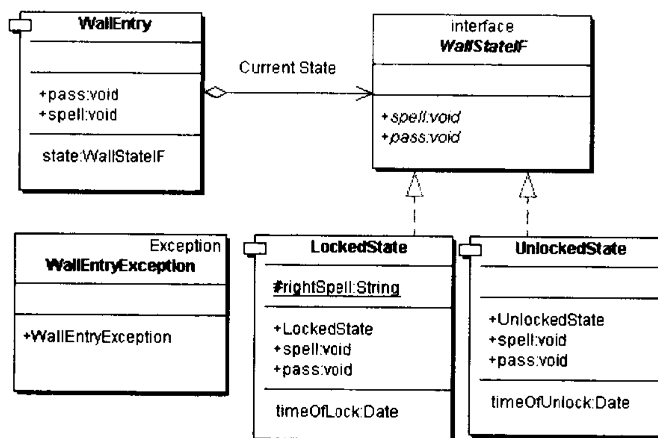
乃以箸掷月中。见一美人，自光中出。初不盈尺，至地遂与人等。纤腰秀项，翩翩作“霓裳舞”。已而歌曰：“仙仙乎，而还乎，而幽我于广寒乎！”其声清越，烈如箫管。歌毕，盘旋而起，跃登几上，惊顾之间，已复为箸。

对于这样复杂的状态和事件，这样嵌套的条件转移可能会持续好几屏幕，而这些条件转移语句中的段落看上去都很相似，仅有一些量值的置换。更重要的是，这样的嵌套语句把状态、状态转移的判断逻辑和对象在不同状态下的行为都混合在一起处理，使得维护工作变得困难，比如老道的法术再多变出一些花样，修改条件转移语句就很困难。

因此，这种条件转移式的处理方法不仅仅是不够美观，它违背了责任分割包装的面向对象的设计理念，要在代码维护和系统扩充上付出代价。

状态模式的处理方法

面向对象的设计原则是分割不同的责任以委派给不同的对象。这里要解决的责任分割和责任委派便是行为责任和做状态判断的逻辑责任的分割和委派。依照状态模式，重新做出如下图所示的设计。



在上图可以看出，系统有如下的角色：

- 环境角色：由 WallEntry 扮演。此类定义了为客户端感兴趣的接口，并持有一个具体状态类的实例，以定义它当前所处的状态。
- 状态角色：由 WallStateIF 扮演。此类定义了把依赖于一个特定的状态所对应的行为包装起来所需要的接口。

- 具体状态角色：由 LockedState、UnlockedState 扮演。它们中的每一个都实现了环境类的状态所对应的行为。



对象在不同状态下的行为包装到不同的状态类里面，这是状态模式所要求的。状态模式所不指明的是控制状态转移的逻辑放在那里。在实现时，这种逻辑可以放在两个地方：一是放在环境类里面，二是包装在状态类里面。这里选择把状态转移的逻辑放在状态类里面。状态对象自己决定何时进行状态转移，以及下一个状态是谁。

责任的分割和委派就是这样实现的，请看下面的源代码。

首先，这里需要定义一个异常类，如代码清单 2 所示，以供后面使用。

代码清单 2：异常类 WallEntryException 的源代码

```
public class WallEntryException extends Exception
{
    public WallEntryException(String message)
    {
        super(message);
        System.out.println(message);
    }
}
```

接口 WallStateIF 的源代码如代码清单 3 所示。

代码清单 3：接口 WallStateIF 的源代码

```
public interface WallStateIF
{
    void spell(WallEntry wall, String spell) throws WallEntryException;
    void pass(WallEntry wall) throws WallEntryException;
}
```

这个接口规范所有的状态类的行为，下面是环境类 WallEntry 的源代码，如代码清单 4 所示。

代码清单 4：环境类 WallEntry 的源代码

```
public class WallEntry
{
    public void setState(WallStateIF state)
    {
        this.state = state;
    }
    public void pass() throws WallEntryException
    {
        state.pass(this);
    }
```

```
    public void spell(String spell)throws WallEntryException
    {
        state.spell(this, spell);
    }
    private WallStateIF state;
}
```

下面给出具体状态类 LockedState 的源代码，如代码清单 5 所示。

代码清单 5：具体状态类 LockedState 的源代码

```
import java.util.Date;
public class LockedState implements WallStateIF
{
    public LockedState()
    {
        this.setTimeOfLock(new Date());
    }
    public void spell(WallEntry wall, String spell) throws WallEntryException
    {
        if (spell.equals(this.rightSpell))
        {
            wall.setState(new UnlockedState());
        }
        else
        {
            throw new WallEntryException("Wrong spell!!!");
        }
    }
    public void pass(WallEntry wall)  throws WallEntryException
    {
        throw new WallEntryException("Bang!!!");
    }
    public Date getTimeOfLock()
    {
        return timeOfLock;
    }
    public void setTimeOfLock(Date timeOfLock)
    {
        this.timeOfLock = timeOfLock;
    }
    private Date timeOfLock;
    protected static String rightSpell = "天机不能泄漏";
}
```

这个具体状态类包装了墙在“闭”状态时的行为。当调用 spell()方法时，此方法可以根据所念的口诀是否正确来决定是否将墙的状态改为“开”状态。当口诀念的不对时，就会给出 WallEntryException 异常。当调用 pass()方法时，状态类所给出的行为是“砰”，给

出 WallEntryException 异常，如代码清单 6 所示。

代码清单 6：具体状态类 UnlockedState 的源代码

```
import java.util.Date;
public class UnlockedState implements WallStateIF
{
    public UnlockedState()
    {
        this.setTimeOfUnlock(new Date());
    }
    public void spell(WallEntry wall, String spell) throws WallEntryException
    {
    }
    public void pass(WallEntry wall) throws WallEntryException
    {
        wall.setState(new LockedState());
    }
    public Date getTimeOfUnlock()
    {
        return timeOfUnlock;
    }
    public void setTimeOfUnlock(Date timeOfUnlock)
    {
        this.timeOfUnlock = timeOfUnlock;
    }
    private Date timeOfUnlock;
```

这个具体状态类包装了墙在“开”状态时的行为。

参考文献

[JZ91]R. E. Johnson and J. Zweig. Delegation in C++. Journal of Object-Oriented Programming，4(11)：22-35，November 1991

第 52 章　专题：单分派和多分派

本章为本书的“访问者（Visitor）模式”一章提供准备知识。

52.1　分派的概念

变量被声明时的类型叫做变量的静态类型（Static Type），有些作者又把静态类型叫做明显类型（Apparent Type）；而变量所引用的对象的真实类型又叫做变量的实际类型（Actual Type）。

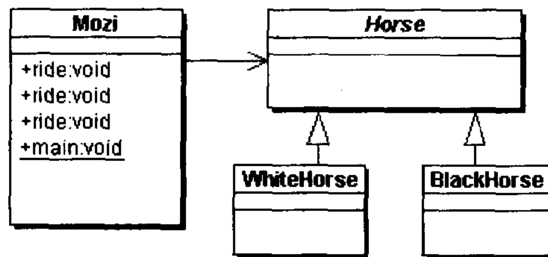
这种根据对象的类型而对方法进行的选择，就是分派（Dispatch）。分派（Dispatch）是面向对象的语言所提供的关键特性之一。根据分派发生的时期，可以将分派分为两种，即静态分派和动态分派。

静态分派（Static Dispatch）发生在编译时期，分派根据静态类型信息发生。静态分派对于读者来说应当并不陌生，方法重载（Overloading）就是静态分派。

动态分派（Dynamic Dispatch）发生在运行时期，动态分派动态地置换掉某个方法。面向对象的语言利用动态分派来实现方法置换产生的多态性。

方法重载

Java 通过方法重载支持静态分派。作为一个例子，考察一下墨子骑马的故事。墨子可以骑白马或者黑马。墨子与白马、黑马和马的类图如下图所示。



在这个模拟系统中，墨子由 Mozi 类代表，它的源代码如代码清单 1 所示。

代码清单 1：Mozi 类的源代码

```
package com.javapatterns.doubledispatch.overload;
public class Mozi
{
    public void ride(Horse h)
    {
```

```
            System.out.println("Riding a horse");
        }
        public void ride(WhiteHorse wh)
        {
            System.out.println("Riding a white horse");
        }
        public void ride(BlackHorse bh)
        {
            System.out.println("Riding a black horse");
        }
        public static void main(String[] args)
        {
            Horse wh = new WhiteHorse();
            Horse bh = new BlackHorse();
            Mozi mozi = new Mozi();
            mozi.ride(wh);
            mozi.ride(bh);
        }
    }
```

显然，Mozi 类的 ride()方法是由三个方法重载而成的。这三个方法分别接受马（Horse）、白马（WhiteHorse）、黑马（BlackHorse）等类型的参量；而马、白马、黑马组成马的等级结构（参见前面的类图）。

那么在运行时，程序会打印出什么结果呢？结果是程序会打印出相同的两行"Riding a horse"。换言之，墨子发现他所骑的都是马。

为什么呢？两次对 ride()方法的调用传入的是不同的参量，也就是 wh 和 bh。它们虽然具有不同的真实类型，但是它们的静态类型都是一样的，均是 Horse 类型。

重载方法的分派是根据静态类型进行的，这个分派过程在编译时期就完成了[BLOCH01]，这就是静态分派的例子。

动态分派

Java 通过方法的置换（Overriding）支持动态分派。比如，考察下面的代码[LISKOV01]，如代码清单 2 所示。

代码清单 2：一个动态单分派的例子

```
String s1 = "ab";
Object o = s1 + "c";

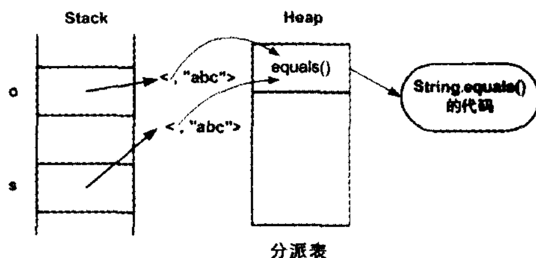
String s = "abc";

boolean b = o.equals(s);
```

变量 s1 和 s 的静态类型和真实类型都是 String，而 o 的静态类型是 Object，真实类型则是 Object 类型的一个子类型。

如果上面最后一行的 equals()方法调用的是 String 类的 equals()方法，那么上面代码检查的就是 o 的值是否等于字符串"abc"；相反，如果上面的 equals()方法调用的是 Object 类的 equals()方法，那么检查的就是 o 所指的对象和 s1 所指的对象是不是同一个对象。

所以，问题的核心是 Java 编译器在编译时期并不总是知道哪一些代码会被执行，因为编译器仅仅知道对象的静态类型，而不知道对象的真实类型；而方法的调用则是根据对象的真实类型，而不是静态类型。仍以上面的代码为例，变量 o 指向一个类型为 String 的对象，这个 String 对象的值是"abc"。这样一来，上面最后一行的 equals()方法调用的是 String 类的 equals()方法，而不是 Object 类的 equals()方法，如右图所示。



上面的类图描述了分派的一种工作方式：每一个对象都持有一个对分派表的引用，对象的每一个方法都在表中有一行信息，而编译器会生成代码访问表中指明的代码的位置，并在调用此方法时，执行这些代码。

在上面的图中，对象 o 调用 equals()方法时，分派表将调用分派给 String 类所实现的 equals()代码，并执行这些代码。所以，最后 b 的值为 true。

分派的类型

一个方法所属的对象叫做方法的接收者，方法的接收者与方法的参量统称做方法的宗量。

根据分派可以基于多少种宗量，可以将面向对象的语言划分为单分派（Uni-Dispatch）语言和多分派（Multi-Dispatch）语言。单分派语言根据一个宗量的类型进行对方法的选择，多分派语言根据多于一个的宗量的类型对方法进行选择。

由于有些文献并不区分多重分派（Multiple Dispatch）与多分派（Multi-Dispatch），因此，本书提醒读者注意两者的区别：多重分派是由一系列的单分派组成的分派过程；而多分派则是不能分解成为多个单分派的分派过程。

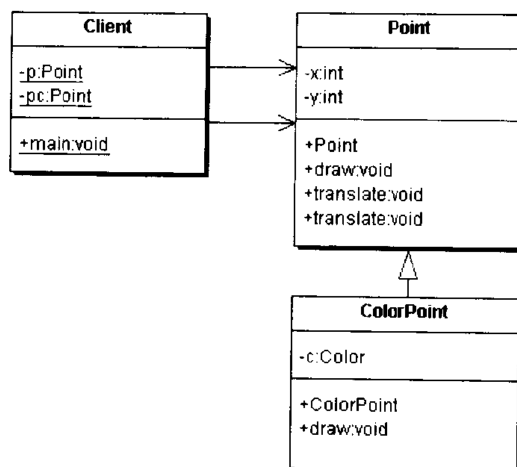
C++和 Java 以及 Smalltalk 均是单分派语言；多分派语言的例子包括 CLOS 和 Cecil。按照这样的区分，C++和 Java 就是动态的单分派语言，因为这两种语言的动态分派仅仅会考虑到方法的接收者的类型，同时又是静态的多分派语言，因为这两种语言对重载方法的分派会考虑到方法的接收者的类型以及方法的所有参量的类型。

在一个支持动态单分派的语言里面，有两个条件决定了一个请求会调用哪一个操作：一是请求的名字，二是接收者的真实类型。单分派限制了方法的选择过程，使得只有一个宗量可以被考虑到，这个宗量通常就是方法的接收者。在 Java 语言里面，如果一个操作是作用于某个类型不明的对象上面的，那么对这个对象的真实类型测试仅会发生一次，这就是动态的单分派的特征。

一言以蔽之，Java 语言支持静态的多分派和动态的单分派。

一个静态和动态分派的例子

下面的例子说明了在 Java 语言中静态多分派和动态单分派是如何发生的。这个例子涉及到一个基类 Point、它的子类 ColorPoint 和使用它们的客户类 Client，如下图所示。



首先来看一看基类 Point 的源代码，如代码清单 3 所示，这个类有两个方法：draw(Canvas)和 translate(int)。

代码清单 3：基类 Point 的源代码

```
public class Point
{
    int x, y;

    void draw(Canvas c)
    {
        //write you code here
    }

    void translate(int d)
    {
        x += d;
        y += d;
    }
}
```

然后看一看子类 ColorPoint 的源代码，如代码清单 4 所示，这个子类置换掉了父类的 draw(Canvas)方法。

代码清单 4：子类 ColorPoint 的源代码

```
public class ColorPoint
{
    Color c;
```



```
    void draw(Canvas C)
    {
        //Color point code
    }
}
```

客户类 Client 声明了两个 Point 类型的变量。在 main()方法里面，这两个同为 Point 类型的变量分别指向 Point 类型和 ColorPoint 类型的对象，因此，这两个变量的实际类型便分别是 Point 和 ColorPoint，如代码清单 5 所示。

代码清单 5：客户端的源代码

```
public class Client
{
    private static Point p ;
    private static Ponit Pc;
    public static void main(String[] args)
    {
        p = new Point();
        pc = new ColorPoint();
        //静态多分派
        p.translate(5);       // 调用接收一个 int 类型参量的版本
        p.trasnlate(1,2);     // 调用接收两个 int 类型参量的版本
        //动态单分派
        p.draw(aCanvas);      //调用 Point.draw()
        pc.draw(aCanvas);     //调用 ColorPoint.draw();
    }
}
```

从客户类可以看出，由于方法的重载，对 Point 对象的两种 translate()方法的调用是典型的静态多分派；而对子类以及父类的 draw()方法的调用则是典型的动态单分派。

（本例子参考了文献[COOTS01]中的一个例子，本书鼓励读者阅读原文）

52.2 双 重 分 派

一个方法根据两个宗量的类型来决定执行不同的代码，这就是“双分派”或者“双重分派”。Java 语言不支持动态的多分派，也就意味着 Java 不支持动态的双分派。但是通过使用设计模式，也可以在 Java 语言里实现动态的双重分派。

双重分派是多重分派的一种，双重分派是由连续的两次单分派组成的。那样怎么在 Java 语言中实现动态的双重分派呢？

第一个方案：类型判断

读者不难想到，既然一个方法需要根据两个宗量判断所执行的代码，那么为什么不将

两个宗量当做方法的参量传进方法中呢？在方法里面可以使用 instanceof 这样的运行时期类型判断来断定这两个参量的动态类型吗？

这当然是可以的。而且，由于 Java 语言支持动态的单分派，也就是说 Java 语言会自动根据方法的接收者进行一次动态单分派。那么实际上为了完成双重分派，只需要将第二个宗量当做方法的参量传入方法中就可以了，而不需要将两个宗量都作为方法的参量。

实际上，有经验的 Java 设计师可能都已经这样做过。Java 语言的 API 中也有很多这样的例子。比如 Component 类中处理事件的方法 processEvent()就是这样的。Java 语言必须根据下面的这两个宗量决定执行哪一段代码：

（1）方法的接收者。换言之，就是当前的 Component 的具体子类。

（2）事件对象，即一个类行为 Event 的对象。由于不同的事件会造成完全不同的行为，因此，事件的类型也就是 Event 的具体子类型，会影响到 processEvent()方法的反应。

换言之，在 java.awt.Component 中，方法 processEvent()的宗量类型是 Component 和 Event。这两者都是基类型，但是这个方法却需要针对宗量的不同子类型采取不同的行动。

不妨看一看下面这一段源代码[COOTS01]，如代码清单 6 所示。

代码清单 6：java.awt.Component 中双重分派的一个例子

```
public abstract class Component implements
    ImageObserver, MenuContainer, Serializable
{
    ......
    protected void processEvent(AWTEvent e)
    {
        if (e instanceof FocusEvent)
        {
            processFocusEvent((FocusEvent)e);
        }
        else if (e instanceof MouseEvent)
        {
            switch(e.getID())
            {
              case MouseEvent.MOUSE_PRESSED:
              case MouseEvent.MOUSE_RELEASED:
              case MouseEvent.MOUSE_CLICKED:
              case MouseEvent.MOUSE_ENTERED:
              case MouseEvent.MOUSE_EXITED:
                processMouseEvent((MouseEvent)e);
                break;
              case MouseEvent.MOUSE_MOVED:
              case MouseEvent.MOUSE_DRAGGED:
                processMouseMotionEvent((MouseEvent)e);
                break;
            }
        }
        else if (e instanceof KeyEvent)
```

```
            {
                processKeyEvent((KeyEvent)e);
            }
            else if (e instanceof ComponentEvent)
            {
                processComponentEvent((ComponentEvent)e);
            }
            else if (e instanceof InputMethodEvent)
            {
                processInputMethodEvent((InputMethodEvent)e);
            }
            else if (e instanceof HierarchyEvent)
            {
            switch (e.getID())
            {
              case HierarchyEvent.HIERARCHY_CHANGED:
                processHierarchyEvent((HierarchyEvent)e);
                break;
              case HierarchyEvent.ANCESTOR_MOVED:
              case HierarchyEvent.ANCESTOR_RESIZED:
                processHierarchyBoundsEvent((HierarchyEvent)e);
                break;
            }
        }
        ......
    }
```

当一个方法明显地检查一个宗量的类型，并根据类型的不同而执行不同的代码时，双重分派就发生了。

可以看出，在上面的 java.awt.Component 源代码中，processEvent(AWTEvent)方法必须按照事件对象所属类型以不同的方式处理事件。因为所有的事件都处于同一个事件队列中，编译器没有动态的类型信息。当队列中的一个元素被取出处理时，程序必须检查其动态类型，以便针对不同的类型做不同的处理。

使用这种办法实现的双重分派程序都格外的冗长、复杂和容易出错。加入一个新的类型不仅需要修改处理事件的代码，而且需要修改相应的条件转移语句，这不符合“开–闭”原则的要求。

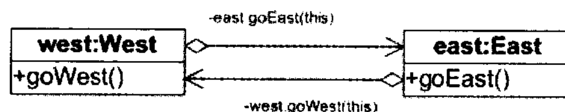
第二个方案：“返传球”

由于第一种方案有不可避免的缺点，因此，必须继续寻找更为灵活的双重分派设计。

“返传球”

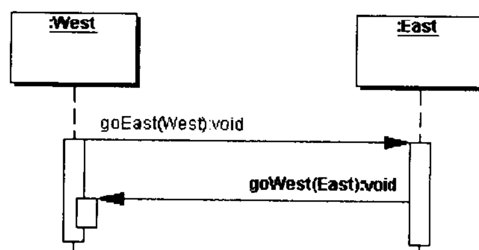
其实只要仔细想一想，读者应该可以想到，既然 Java 语言支持动态的单分派，那么为什么不可以通过两次方法调用来达到两次分派的目的呢？通过两次调用达到两次分派的

目的的类图如下图所示。



在图中有两个对象，为了简单起见，左边的一个叫做 west，右边的那个叫做 east，其类型分别是 West 和 East。这两个类分别有两个方法，叫做 goWest()和 goEast()。

现在 west 对象首先调用 east 对象的 goEast()方法，并将它自己传入。在 east 对象被调用时，立即根据传入的参量知道了调用者是谁，于是反过来调用调用者对象的 goWest()方法。通过两次调用将程序控制权轮番交给两个对象，其时序图如下图所示。

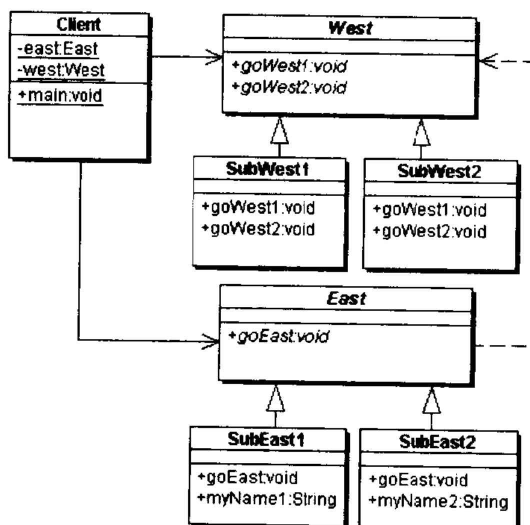


这样就出现了两次方法调用，程序控制权被两个对象像传球一样，首先由 west 对象传给了 east 对象，然后又被返传给了 west 对象。

方法的置换

仅仅是返传了一下球，并不能解决双重分派的问题。关键是怎样利用这两次调用，以及 Java 语言的动态单分派功能，使得在这种传球的过程中，能够触发两次（动态）单分派。

根据本章前面的讨论可以看出，动态单分派在 Java 语言中是在子类型的方法将父类型的方法置换掉时发生的。换言之，West 和 East 都必须分别置身于自己的类型等级结构中，如下图所示。



在这个系统里面，有一个客户端角色 Client，以及两个等级结构——West 等级结构和 East 等级结构。在 West 等级结构中，由一个抽象类 West 定义出公共的类型，并有两个具体子类实现了抽象类的接口；而 East 等级结构也与此相像，由一个抽象类 East 定义出公共的类型，并有两个具体子类实现了这个抽象类的接口。

源代码

为了说明这个系统是怎么工作的，特地给出一部分源代码。

首先是 West 等级结构中的具体类 SubWest1 和 SubWest2 的源代码，如代码清单 7 和代码清单 8 所示。

代码清单 7：SubWest1 类的源代码

```
package com.javapatterns.doubledispatch.ballkicking1;
public class SubWest1 extends West
{
    public void goWest1(SubEast1 east)
    {
        System.out.println( "SubWest1 + " + east.myName1() );
    }
    public void goWest2(SubEast2 east)
    {
        System.out.println( "SubWest1 + " + east.myName2() );
    }
}
```

代码清单 8：SubWest2 类的源代码

```
package com.javapatterns.doubledispatch.ballkicking1;
public class SubWest2 extends West
{
    public void goWest1(SubEast1 east)
    {
        System.out.println( "SubWest2 + " + east.myName1() );
    }
    public void goWest2(SubEast2 east)
    {
        System.out.println( "SubWest2 + " + east.myName2() );
    }
}
```

可以看出，West 的具体子类为不同的 East 具体类型配备了两个不同的方法。East 等级结构中的具体类 SubEast1 和 SubEast2 的源代码如代码清单 9 和代码清单 10 所示。

代码清单 9：SubEast1 类的源代码

```
package com.javapatterns.doubledispatch.ballkicking1;
public class SubEast1 extends East
{
    public void goEast(West west)
```

```
    {
        west.goWest1(this);
    }
    public String myName1()
    {
        return "SubEast1";
    }
}
```

代码清单 10：SubEast2 类的源代码

```
package com.javapatterns.doubledispatch.ballkicking1;
public class SubEast2 extends East
{
    public void goEast(West west)
    {
        west.goWest2(this);
    }
    public String myName2()
    {
        return "SubEast2";
    }
}
```

可以看出，East 类型的 goEast()方法是接受抽象类型的，因此，也可以接受任何具体的 East 类型。

最后是客户端的源代码，如代码清单 11 所示。

代码清单 11：客户端的源代码

```
package com.javapatterns.doubledispatch.ballkicking1;
public class Client
{
    private static East east;
    private static West west;
    public static void main(String[] args)
    {
        // 组合 1
        east = new SubEast1();
        west = new SubWest1();
        east.goEast(west);
        // 组合 2
        east = new SubEast1();
        west = new SubWest2();
        east.goEast(west);
    }
}
```

这里把两个抽象类型的源代码省略了。



各个 West 类中出现了两个方法 goWest1()和 goWest1()，它们的区别在于参量的不同。在这个时候，可以将这两个方法重载，使用同一个名字。这里使用两个不同的名字，是为了强调本系统所实现的双重分派机制是与方法的重载无关的。

此外，East 的具体子类中分别有 myName1()和 myName2()方法，这两个方法名字不同，是为了显示这个系统可以处理具有不同商业方法的 East 类。

活动时序

下面考察一下客户端中组合 1 部分的代码是如何执行的。

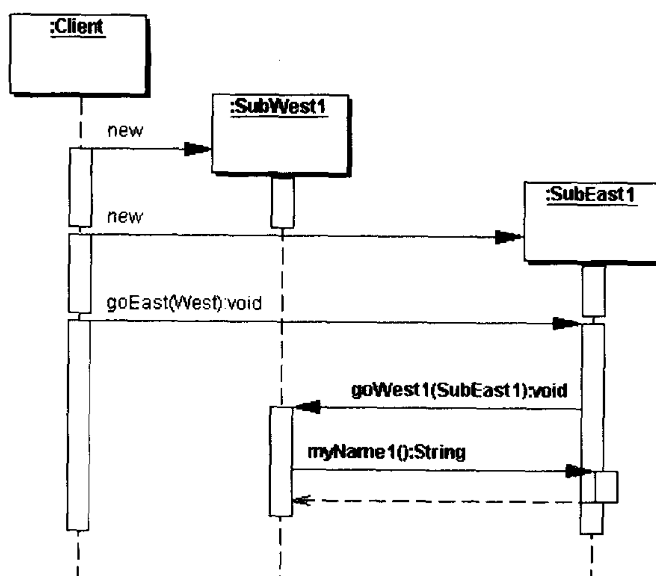
系统在运行时，会首先创建 SubWest1 和 SubEast1 对象，然后客户端调用 SubEast1 的 goEast()方法，并将 SubWest1 对象传入。



由于 SubEast1 对象置换了其超类 East 的 goEast()方法，因此，这个时候就发生了一次动态的单分派。

当 SubEast1 对象接到调用时，会从参数中得到 SubWest1 对象，所以它就立即调用这个对象的 goWest1()方法，并将自己传入。由于 SubEast1 对象有权选择调用哪一个对象，因此，在此时又进行一次动态的方法分派。

这个时候 SubWest1 对象就得到了 SubEast1 对象。通过调用这个对象 myName1()方法，就可以打印出自己的名字和 SubEast 对象的名字，其时序图如下图所示。



系统运行的结果如代码清单 12 所示，其中第一行是组合 1 的运行结果。

代码清单 12：系统运行的结果

```
SubWest1 + SubEast1
SubWest2 + SubEast1
```

由于这两个名字一个来自 East 等级结构，另一个来自 West 等级结构中，因此，它们

的组合式是动态决定的。这就是动态双重分派的实现机制。

最后需要指出的是，首先 East 和 West 等级结构的大小是彼此独立的。虽然在上面的例子中两者都包含了两个具体成员，但这仅是巧合而已；其次 West 要访问的 East 等级结构中的节点数目应当与 West 对象所配备的方法的数目相等，这是必然的。

访问者模式

其实这种返传球的设计就是访问者模式[GOF95]的精华。关于访问者模式，请读者阅读本书的“访问者（Visitor）模式”一章。

52.3 未来的 Java 语言

有经验的读者会认识到，一个稍微复杂的 Java 系统都明显地需要动态双重分派的功能。那么未来的 Java 语言是否会直接支持动态的多分派呢？

实际上，现在人们提出了很多种方案，可以通过对 Java 语言的改造，使 Java 语言实现动态的多分派。比如，下面就是一种叫做 Tuple 的方案，这个方案允许每一个方法能有多于一个的接收者，如代码清单 13 所示。

代码清单 13：一种 Java 语言的改进方案

```
<c1,c2>.method(param1, param2)
```

由于上面的这个方法是属于对象 c1 和 c2 的，因此，上面的这个调用最终执行的方法取决于 c1 和 c2 两者的类型。

与多分派同时存在的当然是对多态性的推广，这种推广后的多态性叫做多多态性（Multi-Polymorphism）。

但是，多分派机制的普遍问题是效率低下、逻辑复杂而且不易学习。因此，Java 语言是否应当支持双分派，是一个颇有争议的问题。

目前，设计师可以通过使用设计模式在 Java 语言中实现双重分派。

问答题

1. 请给出在用户界面设计中常常需要双重分派的一个例子。
2. 请给出 Java 语言 API 中使用静态双分派的一个例子。
3. 请问在本书给出的所有设计模式中，除了访问者模式之外，还有哪一个模式可以给出多重分派的功能。
4. 请给出一个示意性实现，做到混合双重分派，也就是一次静态分派加上一次动态分派。
5. 请简单地描述一下动态三重分派的实现办法。

问答题答案

1. 一个需要使用双重分派的经典例子，就是利用拖放（drag-and-drop）原理设计用户界面的应用软件系统。在支持拖放时，用户行为的结果不仅仅取决于被拖（drag）的数据对象，而且取决于放（drop）的目标对象。如果设计师想要支持不同的数据对象被拖放到不同的目标对象上的情况，就不得不将操作重新分派到更加具体的方法。

这就是说，在这里需要双重分派。

2. 所谓双分派，就是根据方法的接收者以及某一个参量的类型的不同而执行不同的代码。



双分派与双重分派不同，双分派是不能分解成两次分派的。

由于 Java 语言以方法重载的方式支持静态的多分派，因此，只要能找到一个方法重载的例子就满足要求了，而方法重载在 Java 语言的 API 中到处都是。

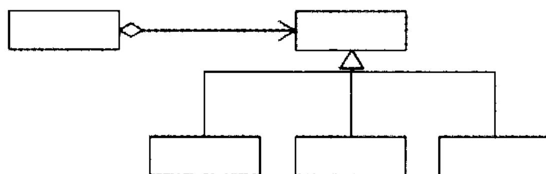
比如 Comparable 接口中的 compareTo()方法。下面就是 java.util.Date 类中实现这个方法的源代码，如代码清单 14 所示。

代码清单 14：静态双分派的一个例子

```
public int compareTo(Object o)
{
    return compareTo((Date)o);
}
public int compareTo(Date anotherDate)
{
    long thisTime = this.getTime();
    long anotherTime = anotherDate.getTime();
    return (thisTime < anotherTime ? -1 : (thisTime==anotherTime ? 0 : 1));
}
```

可以看出，根据方法的接收者不同，以及传入的参量的类型的不同，程序所执行的 compareTo()方法是不同的。这就是静态的双分派。

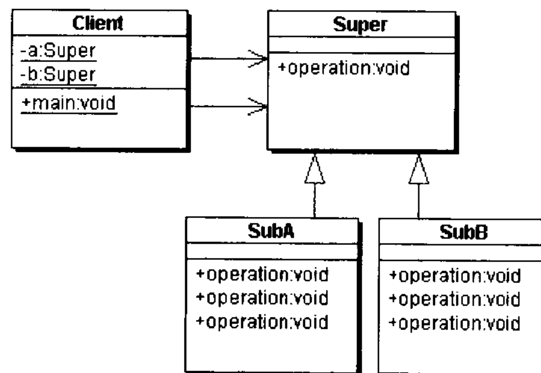
3. 在本书所给出的二十几种设计模式中，除了访问者模式之外，策略模式也可以提供多重分派的功能。策略模式的简略类图如下图所示。



实际上，策略模式可以根据任何的信息当然也包括对象的类型信息，选择提供某一个算法或者行为。

4. 由于 Java 语言通过方法的重载（Overload）支持静态多分派，而通过方法的置换（Override）支持动态的单分派。因此，要做到混合的双重分派，可以通过设计一个方法的重载加上一个方法的置换做到。

在如下图所示的系统中，就实现了方法的重载(请见子类 SubA 和 SubB 类中 operation()方法的重载)，以及方法的置换（超类 Super 的方法 operation()被子类的方法所置换）。



超类的源代码如代码清单 15 所示。

代码清单 15：超类的源代码

```
package com.javapatterns.doubledispatch.mixed;
public class Super
{
    public void operation(Super s)
    {
        System.out.println("This is Super.operation(Super);");
        s.operation(this);
    }
}
```

子类 SubA 的源代码如代码清单 16 所示。

代码清单 16：子类的源代码

```
package com.javapatterns.doubledispatch.mixed;
public class SubA extends Super
{
    public void operation(SubA a)
    {
        System.out.println("This is SubA.operation(SubA);");
    }
    public void operation(SubB b)
    {
        System.out.println("This is SubA.operation(SubB);");
    }
    public void operation(Super s)
    {
```

```
            System.out.println("This is SubA.operation(Super);");
        }
}
```

子类 SubB 的源代码如代码清单 17 所示。

代码清单 17：子类的源代码

```
package com.javapatterns.doubledispatch.mixed;
public class SubB extends Super
{
    public void operation(SubB b)
    {
        System.out.println("This is SubB.operation(SubB);");
    }
    public void operation(SubA a)
    {
        System.out.println("This is SubB.operation(SubA);");
    }
    public void operation(Super s)
    {
        System.out.println("This is SubA.operation(Super);");
    }
}
```

客户端的源代码如代码清单 18 所示。

代码清单 18：客户端的源代码

```
package com.javapatterns.doubledispatch.mixed;
public class Client
{
    private static Super a;
    private static Super b;
    public static void main(String[] args)
    {
        a = new SubA();
        b = new SubB();
        a.operation(b);
    }
}
```

在运行时，程序会打印出下面的文字，如代码清单 19 所示。

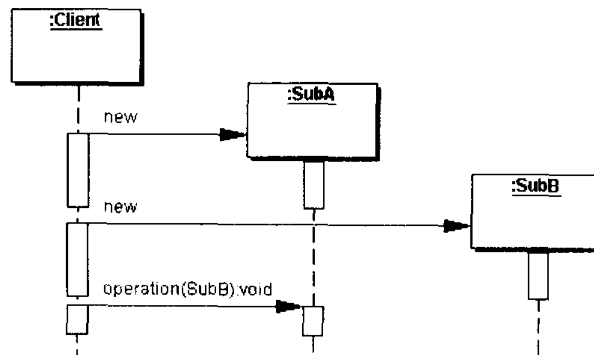
代码清单 19：运行的结果

```
    This is SubA.operation(Super);
```

这表明，a 的静态类型虽然是 Super，但是因为方法的置换是根据真实类型进行的动态分派，因此，SubA.operation()方法被调用，而不是 Super.operation()方法被调用。

但是，由于 SubA.operation()方法是重载的，因此会有第二个分派出现。Java 语言中

的方法重载是静态分派，而静态分派是根据参量的静态类型进行的。因为对象 b 的静态类型是 Super，因此，SubA.operation(Super)方法被调用，从而打印出上面的信息。混合分派的系统时序图如下图所示。



5. 比较一下本章所给出的混合双分派、动态双分派的实现办法就会发现，动态三重分派应当涉及到三个等级结构。

不难发现下面的规律：

- 混合双分派（由一个动态单分派和一个静态单分派组成）：一个等级结构；
- 动态双重分派（由两个动态单分派组成）：两个等级结构；
- 动态三重分派（由三个动态单分派组成）：需要三个等级结构；
- 动态 x 重分派（由 x 个动态单分派组成）：需要 x 个等级结构。

参考文献

[LISKOV01] Barbara Liskov with John Guttag. Program Development in Java–Abstraction，Specification，and Object–Orientd Design. Addison–Wesley，2001.

[MEYERS96] S.Meyers. More Effective C++. Addison–Wesley，New-York，1996

[BLOCH01] Joshua Bloch. Effective Java – Programming Language Guide. published by Addison-Wesley，2001

[COOTS01] Christopher Dutchyn，Paul Lu，Duane Szafron，Steven Bromling，Wade Holst. Multi-Dispatch in the Java Virtual Machine：Design and Implementation，in Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems. San Antonio，Texas，USA ，January 29-February 2，2001

第 53 章　访问者（Visitor）模式

访问者模式是对象的行为模式[GOF95]。访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构则可以保持不变。

53.1　引　　言

建议读者在阅读本书的“合成（Composite）模式”、“迭代子（Iterator）模式”和“专题：单分派和多分派”等章节后，再阅读本章。

问题

聚集是大多数的系统都要处理的一种容器对象，它保存了对其他对象的引用。相信大多数读者都有处理聚集的经验，但是大家处理过的大多数聚集恐怕都是同类对象的聚集。换言之，在聚集上采取的操作都是一些针对同类型对象的同类操作，而迭代子模式就是为这种情况准备的设计模式。

下面就是这样的一个例子，如代码清单 1 所示。

代码清单 1：一个针对聚集的操作

```
public void print(Collection collection)
{
    Iterator iterator = collection.iterator();

    while (iterator.hasNext())
    {
        System.out.println(iterator.next().toString());
    }
}
```

那么一个很多人都没有考虑过的问题就是，如何针对一个保存有不同类型对象的聚集采取某种操作呢？

粗看上去，这并不是什么难题。仍以上面的 print()方法为例，如果 collection 聚集中的元素有可能还是聚集，那么调用聚集的 toString()就没有意义，应该调用它的内部元素的 toString()方法。换言之，上面的方法就应当改写成下面这样，如代码清单 2 所示。

代码清单 2：一个针对聚集的操作

```
public void print(Collection collection)
{
    Iterator iterator = collection.iterator();
    while (iterator.hasNext())
    {
        Object o = iterator.next();
        if (o instanceof Collection)
        {
            print((Collection)o);
        }
        else
        {
            System.out.println(o.toString());
        }
    }
}
```

但是这还没有完，如果这个操作对不同类型的元素有所不同时怎么办呢？比如系统可能会要求在打印字符串时，加上单引号；在打印 Double 类型的数据时，在数据后面加上"D"；在打印 Float 类型的数据时，在数据后面加上"F"。这时，直截了当的答案就是继续修改上面的 print()方法，如代码清单 3 所示。

代码清单 3：聚集的操作细节有所不同

```
public void print(Collection collection)
{
    Iterator iterator = collection.iterator()
    while (iterator.hasNext())
    {
        Object o = iterator.next();
        if (o instanceof Collection)
        {
            print((Collection)o);
        }
        else if (o instanceof String)
        {
            System.out.println("'"+o.toString()+"'");
        }
        else if (o instanceof Double)
        {
            System.out.println(o.toString()+"D");
        }
        else if (o instanceof Float)
        {
            System.out.println(o.toString()+"F");
```

```
            }
            else
            {
                System.out.println(o.toString());
            }
        }
    }
```

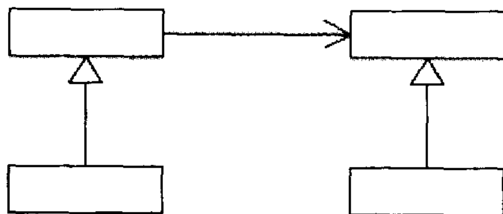
这个条件转移语句变得越来越长，代码也越来越难以维护。

换言之，如果需要针对一个包含不同类型元素的聚集采取某种操作，而操作的细节根据元素的类型不同而有所不同时，就会出现必须对元素类型做类型判断的条件转移语句。阅读过本书的“专题：单分派和多分派”一章的读者会意识到，这就是双重分派的实际应用。

这个时候，使用访问者模式就是一个值得考虑的解决方案。

访问者模式

访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由地演化。访问者模式的简略类图如下图所示。



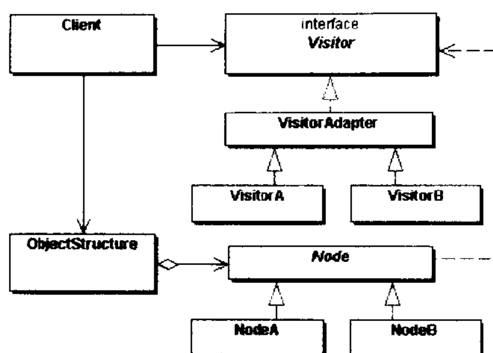
数据结构的每一个节点都可以接受一个访问者的调用，此节点向访问者对象传入节点对象，而访问者对象则反过来执行节点对象的操作。这样的过程叫做“双重分派”。节点调用访问者，将它自己传入，访问者则将某算法针对此节点执行。

双重分派意味着施加于节点之上的操作是基于访问者和节点本身的数据类型，而不仅仅是其中的一者。关于“双重分派”的详细介绍，请见本书的“专题：单分派和多分派”一章。

53.2　访问者模式的结构

如下图所示，这个静态图显示了有两个具体访问者和两个具体节点的访问者模式的设计，必须指出的是，具体访问者的数目与具体节点的数目没有任何关系，虽然在这个示意性的系统里面两者的数目都是两个。





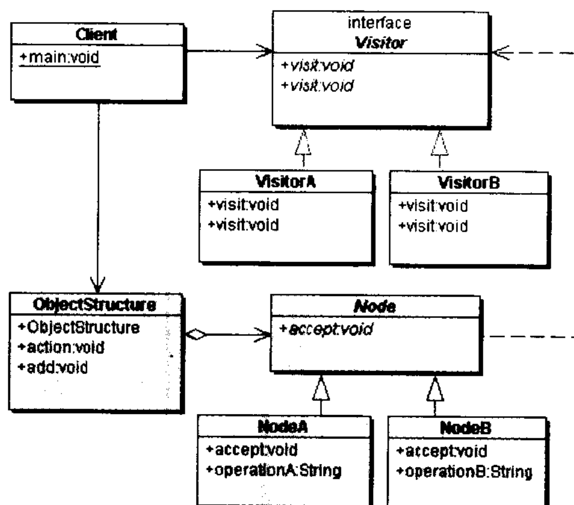
访问者模式所涉及的角色

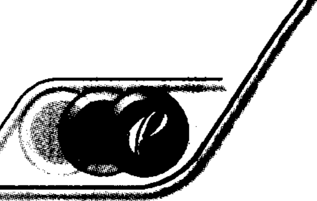
访问者模式涉及到抽象访问者角色、具体访问者角色、抽象节点角色、具体节点角色、结构对象角色以及客户端角色。

- 抽象访问者（Visitor）角色：声明了一个或者多个访问操作，形成所有的具体元素角色必须实现的接口。
- 具体访问者（ConcreteVisitor）角色：实现抽象访问者角色所声明的接口，也就是抽象访问者所声明的各个访问操作。
- 抽象节点（Node）角色：声明一个接受操作，接受一个访问者对象作为一个参量。
- 具体节点（Node）角色：实现了抽象元素所规定的接受操作。
- 结构对象（ObjectStructure）角色：有如下的一些责任，可以遍历结构中的所有元素；如果需要，提供一个高层次的接口让访问者对象可以访问每一个元素；如果需要，可以设计成一个复合对象或者一个聚集，如列（List）或集合（Set）。

类图和示意性源代码

为了能明显看到类的内部细节，特别给出如下图所示的类图。





可以看到，抽象访问者角色为每一个具体节点都准备了一个访问操作。由于有两个节点，因此，对应就有两个访问操作。

抽象访问者角色由一个 Java 接口扮演，它的源代码如代码清单 4 所示。可以看出，抽象访问者角色为每一个节点都提供了一个访问操作，接收相应的节点对象作为参量。

代码清单 4：模式中 Visitor 接口的示意性源代码

```
package com.javapatterns.visitor.visitor0;
public interface Visitor
{
    /**
     *  对应于 NodeA 的访问操作
     */
     void visit(NodeA node);
    /**
     *  对应于 NodeB 的访问操作
     */
     void visit(NodeB node);
}
```

下面是具体访问者 VisitorA 的源代码，如代码清单 5 所示。可以看出，具体访问者 VisitorA 类实现了抽象访问者所声明的两个访问操作。

代码清单 5：模式中具体访问者 VisitorA 类的源代码

```
package com.javapatterns.visitor.visitor0;
public class VisitorA implements Visitor
{
    /**
     *  对应于 NodeA 的访问操作
     */
     public void visit(NodeA nodeA)
     {
          System.out.println( nodeA.operationA() );
     }
    /**
     *  对应于 NodeB 的访问操作
     */
     public void visit(NodeB nodeB)
     {
          System.out.println( nodeB.operationB() );
     }
}
```

这两个访问操作分别对应于系统的两个节点。下面是具体访问者 VisitorB 的源代码，如代码清单 6 所示。

代码清单 6：模式中具体访问者 VisitorB 类的源代码

```
package com.javapatterns.visitor.visitor0;
public class VisitorB implements Visitor
{
    /**
     *  对应于 NodeA 的访问操作
     */
    public void visit(NodeA nodeA)
    {
        System.out.println( nodeA.operationA() );
    }
    /**
     *  对应于 NodeB 的访问操作
     */
    public void visit(NodeB nodeB)
    {
        System.out.println( nodeB.operationB() );
    }
}
```

抽象节点由一个抽象 Java 类扮演，它的源代码如代码清单 7 所示。可以看出，抽象节点声明了一个接受操作。

代码清单 7：模式中的抽象节点 Node 的源代码

```
package com.javapatterns.visitor.visitor0;
abstract public class Node
{
    /**
     *  接受操作
     */
    public abstract void accept(Visitor visitor);
}
```

具体节点 NodeA 类的源代码如代码清单 8 所示。可以看出具体节点声明了一个接受方法，接受一个访问者对象作为参量，这个节点同时还有若干个商业方法，以 operationA() 为代表。

代码清单 8：模式中具体节点 NodeA 类的源代码

```
package com.javapatterns.visitor.visitor0;
public class NodeA extends Node
{
    /**
     *  接受操作
     */
    public void accept(Visitor visitor)
    {
```

```
            visitor.visit(this);
        }
        /**
         *  NodeA 特有的商业方法
         */
        public String operationA()
        {
            return "NodeA is visited";
        }
}
```

与具体节点 NodeA 相似的是具体节点 NodeB，它的源代码如代码清单 9 所示。

代码清单 9：模式中具体节点 NodeB 类的源代码

```
package com.javapatterns.visitor.visitor0;
public class NodeB extends Node
{
    /**
     *  接受操作
     */
    public void accept(Visitor visitor)
    {
        visitor.visit(this);
    }
    /**
     *  NodeB 特有的商业方法
     */
    public String operationB()
    {
        return "NodeB is visited";
    }
}
```

下面是结构对象角色的源代码，如代码清单 10 所示。这个结构对象角色持有一个聚集，并向外界提供 add()方法作为对聚集的管理操作。通过调用这个方法，可以动态地增加一个新的节点。

代码清单 10：模式中结构对象 ObjectStructure 类的源代码

```
package com.javapatterns.visitor.visitor0;
import java.util.Vector;
import java.util.Enumeration;
public class ObjectStructure
{
    private Vector nodes;
    private Node node;
    /**
     *  构造子
```

```
    */
    public ObjectStructure()
    {
        nodes = new Vector();
    }
   /**
    *  执行访问操作
    */
    public void action(Visitor visitor)
    {
        for(Enumeration e = nodes.elements();
            e.hasMoreElements();)
        {
            node = (Node)e.nextElement();
            node.accept(visitor);
        }
    }
   /**
    *  增加一个新的元素
    */
    public void add(Node node)
    {
        nodes.addElement(node);
    }
}
```

下面是客户端角色的源代码，如代码清单 11 所示。

代码清单 11：客户端源代码

```
package com.javapatterns.visitor.visitor0;
public class Client
{
    private static ObjectStructure aObjects;
    private static Visitor visitor;
    static public void main(String[] args)
    {
        // 创建一个结构对象
        aObjects = new ObjectStructure();
        // 给结构增加一个节点
        aObjects.add(new NodeA());
        // 给结构增加一个节点
        aObjects.add(new NodeB());
        // 创建一个新的访问者
        visitor = new VisitorA();
        // 让访问者访问结构
        aObjects.action(visitor);
    }
```



```
}
```

值得指出的是，虽然在这个示意性的实现里并没有出现一个复杂的具有多个树枝节点的对象树结构，但是，在实际系统中访问者模式通常是用来处理复杂的对象树结构的，而且访问者模式可以用来处理跨越多个等级结构的树结构问题。这正是访问者模式的功能强大之处。

对象树结构的设计问题可以采用合成模式处理，请读者参见本书的"合成（Composite）模式"一章。

53.3　系统的时序图

系统的时序图显示了系统的各个角色发生相互作用的时间顺序。在这个时序图中所涉及到的角色有：客户端对象、结构对象、VisitorA 对象、NodeA 对象和 NodeB 对象等。由于过程的复杂性，本节将活动时序划分为两个时序图展示给读者。

准备过程

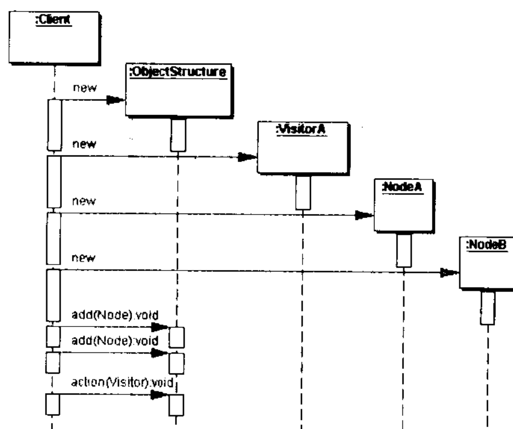
首先，这个示意性的客户端创建了一个结构对象，然后将一个新的 NodeA 对象和一个新的 NodeB 对象传入。

其次，客户端创建了一个 VisitorA 对象，并将此对象传给结构对象。

然后，客户端调用结构对象聚集管理方法，将 NodeA 和 NodeB 节点加入到结构对象中去。

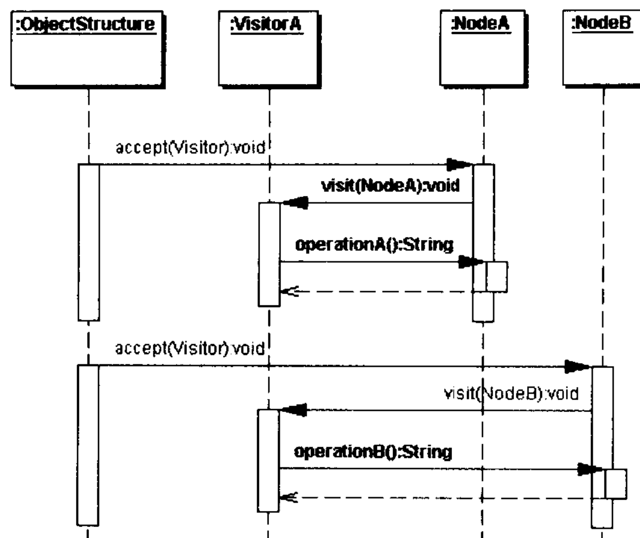
最后，客户端调用结构对象的行动方法 action()，启动访问过程。

客户端对象的时序图如下图所示。



访问过程

如下图所示，这个时序图显示了当结构对象的行动方法被调用时发生的事情。



结构对象会遍历它自己所保存的聚集中的所有节点，在本系统中就是节点 NodeA 和节点 NodeB。首先 NodeA 会被访问到，这个访问是由以下的操作组成的：

（1）NodeA 对象的接受方法被调用，并将 VisitorA 对象本身传入；

（2）NodeA 对象反过来调用 VisitorA 对象的访问方法，并将 NodeA 对象本身传入；

（3）VisitorA 对象调用 NodeA 对象的商业方法 operationA()。

从而就完成了双重分派过程，接着，NodeB 会被访问，这个访问的过程和 NodeA 被访问的过程是一样的，为方便读者，叙述如下：

（1）NodeB 对象的接受方法被调用，并将 VisitorA 对象本身传入；

（2）NodeB 对象反过来调用 VisitorA 对象的访问方法，并将 NodeB 对象本身传入；

（3）VisitorA 对象调用 NodeB 对象的商业方法 operationB()。

从而又完成了一个双重分派过程。

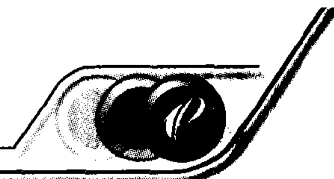
因此，结构对象对聚集元素的遍历过程就是对聚集中所有的节点进行委派的过程，也就是双重分派的过程。换言之，系统有多少个节点就会发生多少个双重分派过程。

53.4 在什么情况下应当使用访问者模式

有意思的是，在很多情况下不使用设计模式反而会得到一个较好的设计。换言之，每一个设计模式都有其不应当使用的情况。访问者模式也有其不应当使用的情况，本节就首先看一看访问者模式不应当在什么情况下使用。

倾斜的可扩展性

访问者模式仅应当在被访问的类结构非常稳定的情况下使用。换言之，系统很少出现需要加入新节点的情况。如果出现需要加入新节点的情况怎么办呢？那时就必须在每一个访问对象里加入一个对应于这个新节点的访问操作，而这是对一个系统的大规模修改，因



而是违背“开-闭”原则的。

访问者模式允许在节点中加入新的方法，相应的仅仅需要在一个新的访问者类中加入此方法，而不需要在每一个访问者类中都加入此方法。

显然，访问者模式提供了倾斜的可扩展性设计：方法集合的可扩展性和类集合的不可扩展性。

换言之，如果系统的数据结构是频繁变化的，则不适合使用访问者模式。

“开-闭”原则和对变化的封装

面向对象的设计原则中最重要的便是所谓的“开-闭”原则。一个软件系统的设计应当尽量做到对扩展开放，对修改关闭。达到这个原则的途径就是遵循“对变化的封装”的原则。这个原则讲的是在进行软件系统的设计时，应当设法找出一个软件系统中会变化的部分，将之封装起来。

很多系统可以按照算法和数据结构分开，也就是说一些对象含有算法，而另一些对象含有数据，接受算法的操作。如果这样的系统有比较稳定的数据结构，又有易于变化的算法的话，使用访问者模式就是比较合适的，因为访问者模式使得算法操作的增加变得容易。

反过来，如果这样一个系统的数据结构对象易于变化，经常要有新的数据对象增加进来的话，就不适合使用访问者模式。因为在访问者模式中增加新的节点很困难，要涉及到在抽象访问者和所有的具体访问者中增加新的方法。

53.5　使用访问者模式的优点和缺点

访问者模式有如下的优点：

（1）访问者模式使得增加新的操作变得很容易。如果一些操作依赖于一个复杂的结构对象的话，那么一般而言，增加新的操作会很复杂。而使用访问者模式，增加新的操作就意味着增加一个新的访问者类，因此，变得很容易。

（2）访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。

（3）访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。迭代子只能访问属于同一个类型等级结构的成员对象，而不能访问属于不同等级结构的对象。访问者模式可以做到这一点。

（4）积累状态。每一个单独的访问者对象都集中了相关的行为，从而也就可以在访问的过程中将执行操作的状态积累在自己内部，而不是分散到很多的节点对象中。这是有益于系统维护的优点。

访问者模式有如下的缺点：

（1）增加新的节点类变得很困难。每增加一个新的节点都意味着要在抽象访问者角色中增加一个新的抽象操作，并在每一个具体访问者类中增加相应的具体操作。

（2）破坏封装。访问者模式要求访问者对象访问并调用每一个节点对象的操作，这

隐含了一个对所有节点对象的要求：它们必须暴露一些自己的操作和内部状态。不然，访问者的访问就变得没有意义。由于访问者对象自己会积累访问操作所需的状态，从而使这些状态不再存储在节点对象中，这也是破坏封装的。

由于显然的缺点，访问者模式成为一个有争议的设计模式。有一些设计师反对使用访问者模式，另外有一些设计师则强调访问者模式的优点，还有一些设计师则千方百计地设法修改访问者模式，克服其不足。事实上，尽管有人反对使用这一模式，访问者模式仍然在很多重要的系统中使用；尽管有人喜欢这一模式，但是其缺点使得它仍然不是一个首选的模式；尽管有一些方案可以增强访问者模式，但是所有的增强方案都使得模式的复杂性增加。

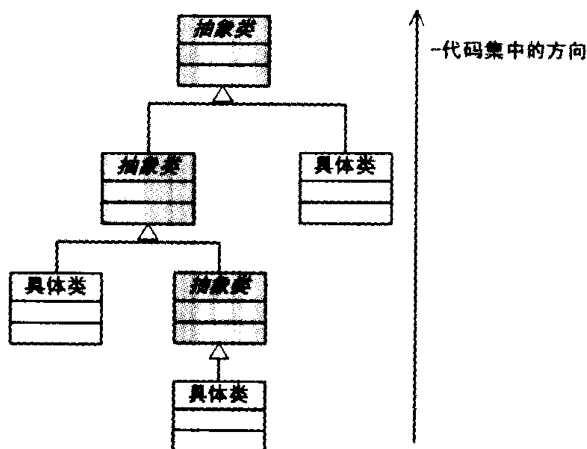
本书将有关访问者模式的事实完整地交代给读者，相信读者可以自行决定自己的观点。

53.6 访问者模式的实现

当实现访问者模式时，要将尽可能多的对象浏览逻辑放在 Visitor 类中，而不是放在它的子类里。这样的话，ConcreteVisitor 类对所访问的对象结构依赖较少，从而使维护较为容易。

对节点结构的优化

访问者模式所涉及的接受访问的结构对象可以属于一个等级结构，也可以属于不同的等级结构。对于每一个等级结构而言，都可以通过将行为向上移动，将状态向下移动而加以优化。代码的优化方向与状态的优化方向恰好相反，如下图所示。



谁负责遍历行为

由于每一个访问者都需要遍历结构对象中的每一个元素，因此，一个自然的问题就是，

哪一个角色负责遍历行为？可供选择的有：结构对象（ObjectStructure）、访问者对象或者创建一个新的迭代对象。

（1）由结构对象负责迭代是最常见的选择，这也就是需要结构对象角色的出发点。

（2）另一个解决方案是使用一个迭代对象来负责遍历行为。

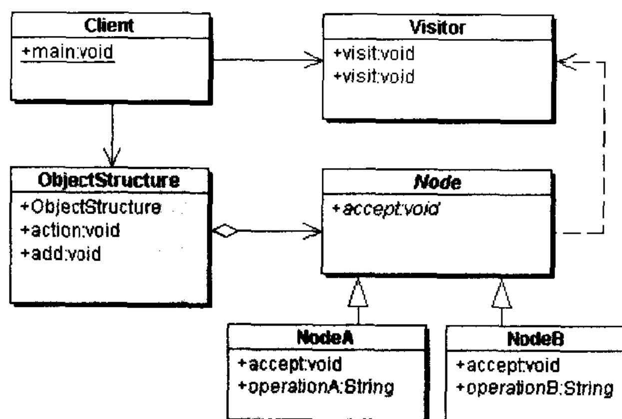
（3）将遍历行为放到访问者中是第三个可能的选择。一般而言，这样做会导致每一个具体访问者都不得不具有管理聚集的内部功能，而这在一般情况下是不理想的。然而如果遍历的逻辑较为复杂的话，将所有的遍历逻辑放到结构对象角色中，不如将逻辑放到各个具体访问者角色中，因此这种做法也不失为一种可行的选择。

是否需要结构对象角色

如果遍历行为是放到具体访问者中的，那么结构对象角色（ObjectStrcture）就可以省略。

是否需要抽象访问者角色

抽象访问者角色的设置是为了将可以复用的部分放在抽象类中，以便具体访问者角色可以继承，从而达到复用的目的。如果设计师非常肯定访问者只有一个，那么设置抽象访问者角色便没有太大意义，完全可以省略。访问者模式的使用如下图所示。



在这种情况下，与前面标准的情况相比，只有访问者角色的源代码需要变化，而其他角色的源代码不需要改变，如代码清单 12 所示。

代码清单 12：惟一的访问者就是这个具体类

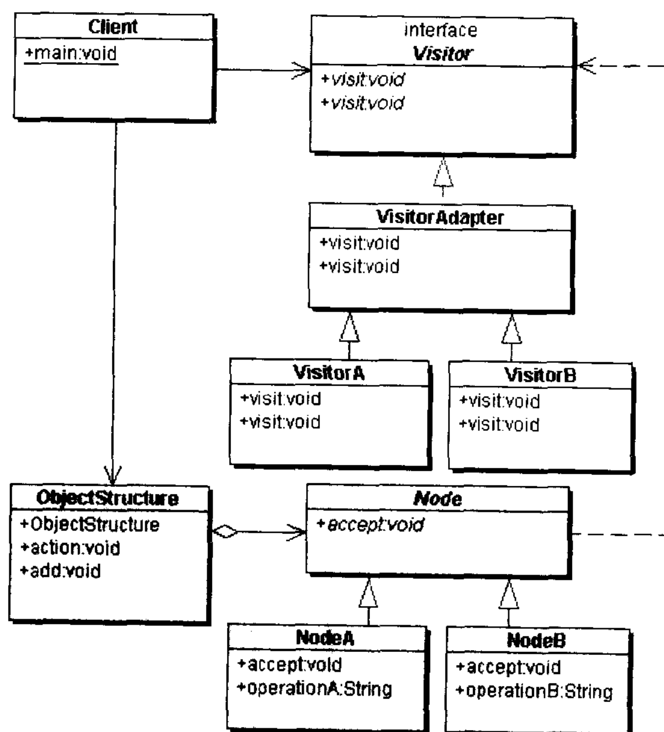
```
package com.javapatterns.visitor.visitorsimplified;
public class Visitor
{
    /**
     *  访问方法
     */
    public void visit(NodeA nodeA)
```

```
    {
        System.out.println( nodeA.operationA() );
    }
   /**
    *  访问方法
    */
    public void visit(NodeB nodeB)
    {
        System.out.println( nodeB.operationB() );
    }
}
```

在这个例子里面，读者可以看到，虽然只有一个访问者，但是因为有两个节点，所以这个访问者仍然需要两个访问操作。

访问者模式与缺省适配模式的联合使用

正如在缺省适配模式一章所指出的那样，缺省适配模式可以适配为两个不同的接口。在这里，缺省适配模式为访问者模式增加另一个层次的继承关系，从而可以利用这个机会为一个多态性的等级结构提供一个默认的实现。访问者模式与适配器模式的联合使用如下图所示。



VisitorAdapter 角色的源代码如代码清单 13 所示。

代码清单 13：Vistor Adapter 类的源代码

```
package com.javapatterns.visitor.visitoradapter;
```

```
public class VisitorAdapter implements Visitor
{
    /**
     *  访问方法
     */
     public void visit(NodeA nodeA){}
    /**
     *  访问方法
     */
     public void visit(NodeB nodeB){}
}
```

具体访问者 VisitorA 的源代码如代码清单 14 所示。

代码清单 14：VistorA 类的源代码

```
package com.javapatterns.visitor.visitoradapter;
public class VisitorA extends VisitorAdapter
{
    /**
     *  访问方法
     */
     public void visit(NodeA nodeA)
     {
          System.out.println( nodeA.operationA() );
     }
    /**
     *  访问方法
     */
     public void visit(NodeB nodeB)
     {
          System.out.println( nodeB.operationB() );
     }
}
```

具体访问者 VisitorB 的源代码如代码清单 15 所示。

代码清单 15：VisitorB 类的源代码

```
package com.javapatterns.visitor.visitoradapter;
public class VisitorB extends VisitorAdapter
{
    /**
     *  访问方法
     */
     public void visit(NodeA nodeA)
     {
          System.out.println( nodeA.operationA() );
     }
```

```
    /**
     *  访问方法
     */
    public void visit(NodeB nodeB)
    {
        System.out.println( nodeB.operationB() );
    }
}
```

这样的一种复合模式常常叫做适配器访问者模式（Adapter Visitor Pattern）。

53.7 电脑专卖系统：问题与对象

本节考察一个电脑专卖店的销售系统设计问题。

问题

这个电脑专卖店既销售单独的电脑零件，也销售组装成集成组件的半成品，而且还销售电脑整机。单独的电脑零件包括机箱（Case）、中心处理器（CPU）、硬盘（Hard Disk）、主机板（Main Board）等；半成品组件包括集成主机板（Integrated Board），也就是已经装上中心处理器的主机板；而成品就是装成整机的电脑，包括机箱、集成主机板、硬盘等。

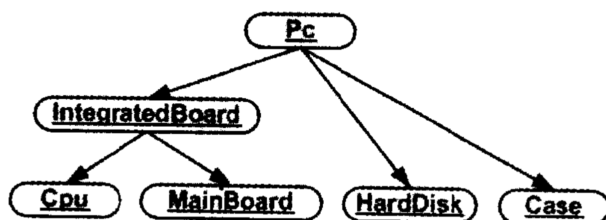
每一种零件都有自己的价格，而本销售系统要做的事情是根据零件的单价对销售单上每一种配置计算出总的价格。由于可能的配置较多，因此，系统需要做的是将每一种配置分解成单独的零件，然后将每一种零件的价格相加，从而得出配置的总价格来。

对象的树结构

系统设计的起点在于对系统所处理的对象的了解。

本系统需要处理的对象就是各种电脑零件和组件对象。这些对象基本上可以分成两种，一种是单纯对象，也就是不能分成更小单位的那些零件对象，包括机箱（Case）、中心处理器（CPU）、硬盘（Hard Disk）、主机板（Main Board）等；另一种则是复合对象，也就是可以分成更小单位的那些组件对象，包括半成品组件和整机电脑。

这样一来，对象的结构就自然形成一个树结构。电脑系统的结构如下图所示。



树图中的节点可以分成两种，一种节点是树枝节点；另一种是树叶节点。为了区别两者，特意在树图中将树枝节点涂上了颜色。可以看到，树枝节点包括 Pc、IntegratedBoard 等，其他节点都是树叶节点。每一个树枝节点都持有对子节点的引用，因此，树图中的连线带有指向子节点的箭头。

合成模式

学习过合成模式的读者到这里就应当想到，合成模式是描述这样的树结构的理想设计模式。本章选择使用安全式的合成模式。

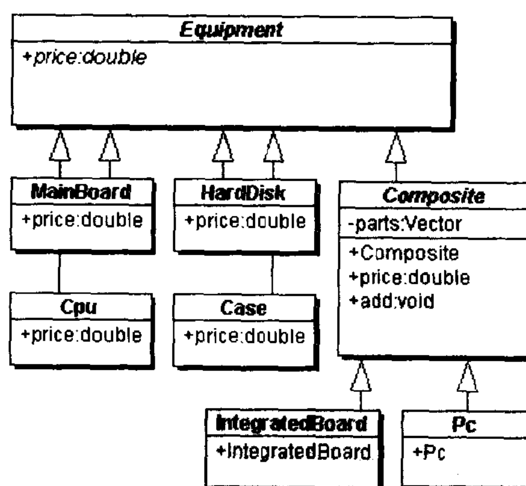
抽象节点角色

使用合成模式描述这样一个树结构，首先需要有一个抽象节点角色，规定出所有的节点都必须实现的接口，包括所有的商业方法。在本系统中，这个抽象节点角色由 Equipment 类扮演。而这个接口只包含一个方法，就是 price()，它给出每一个具体节点的价格。

抽象树枝节点角色

合成模式要求提供一个抽象树枝节点角色，负责规范出所有的具体树枝节点角色的接口。安全式的合成模式要求这个角色还必须提供管理子节点聚集所需的方法。在本系统中，这个角色由 Composite 类扮演，Composite 类给出一个默认的 price()方法，作为所有的成员对象的价格计算方法。另外，Composite 类还提供了一个聚集管理方法 add()，允许客户端增加聚集成员对象。

电脑零部件的结构图如下图所示。



具体树叶节点角色

从上图中读者还可以看到，具体树叶节点对象 Case、Cpu、HardDisk 和 MainBoard 实现了根节点 Equipment 所要求的商业方法 price()。

具体树枝节点角色

而具体树枝节点对象 IntegratedBoard 和 Pc 则继承了 Composite 类所提供的聚集管理

方法——price()方法。它们自己还有非平庸的构造子，在构造子中通过调用超类的聚集管理对象来定义出下级节点的组成。

53.8 电脑专卖系统的访问者模式设计

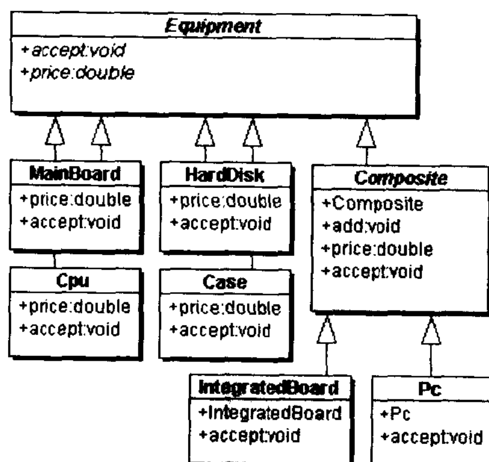
被访问者角色

上面所给出的几乎就是访问者模式所需要的被访问者角色的结构图，只是访问者模式要求这些节点对象必须提供接受方法。



接受方法对于树叶节点和对于树枝节点来说含义不同，树枝节点的访问方法必须委派给所有的子对象。

经过增加这一方法，系统的设计图就变成如下图所示。

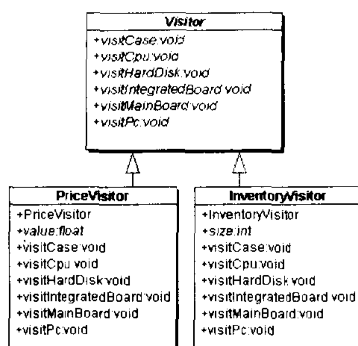


访问者角色

访问者模式的中心在于访问者角色，它们声明并实现访问操作。这些访问操作调用各个被访问者对象的接收操作，构成访问者对象对被访问者对象的访问。

首先，抽象访问者角色提供一个所有的具体访问者角色共有的接口，也就是针对各个被访问者对象的访问方法。每有一个具体节点对象，就对应地有一个访问方法。由于这里有六个具体节点对象，因此一共需要有六个访问方法。抽象访问者角色声明出这六个访问方法，而具体访问者角色则实现这六个访问方法。

除此之外，具体访问者角色还需要给出自己所特有的商业方法。比如，具体访问者 PriceVisitor 类需要一个 value()方法，给出价格的总额；而具体访问者 InventoryVisitor 类则需要一个 size()方法，给出所遍历的树叶节点的总数目，如下图所示。



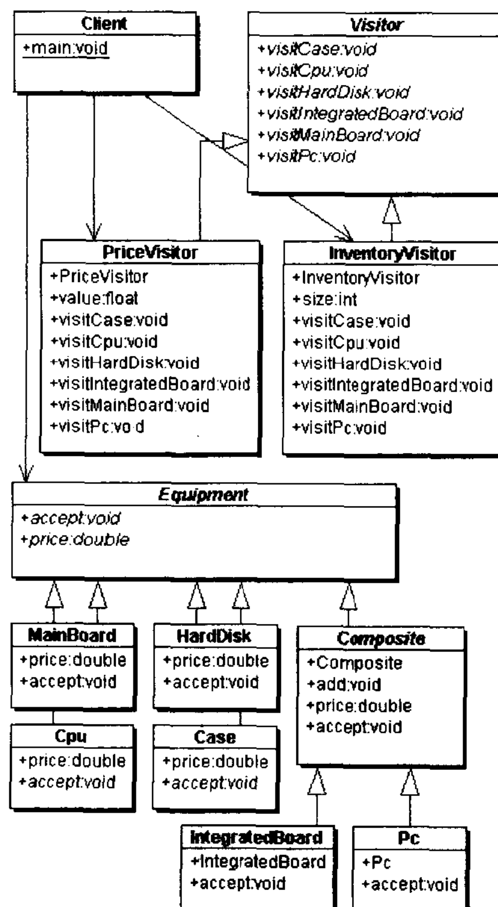
结构对象角色

由于抽象树枝节点角色提供了聚集管理方法，因此，在应用访问者模式的时候，可以省略掉结构对象（ObjectStructure）角色。

这个角色应当有的行动方法也就交给客户端，由客户端直接调用各个具体访问者角色的商业方法实现。

系统设计图

这样一来，不难得出系统的设计总图，如下图所示。



从图中可以看出，这个系统具有两个等级结构，一个是访问者等级结构，以 Visitor 为根节点；另一个是被访问者等级结构，以 Equipment 为根节点。

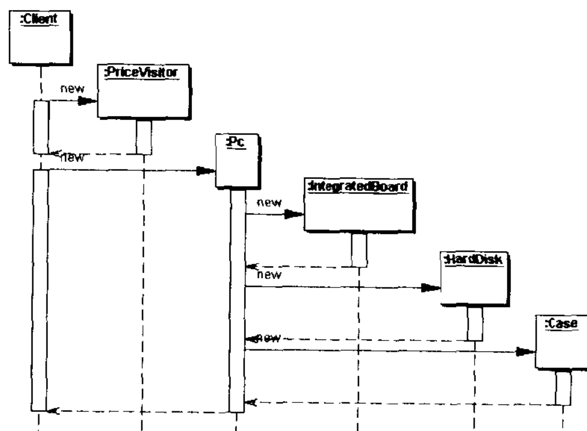
客户端 Client 对象通过调用具体访问者对象的行动方法，包括 PriceVisitor 的 value()方法和 InventorVisitor 的 size()方法，会访问聚集中的所有元素，并同时触发双重分派过程。在遍历过程中，访问者对象会积累状态，这些状态就是客户端需要的数据。

活动时序图

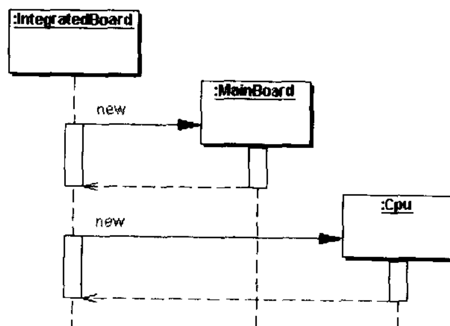
活动时序图反映了系统的各个角色在相互作用时的情况。由于本系统涉及的角色繁多，因此，本节将活动的时序划分成几个时序图，分几个部分讲解 PriceVisitor 角色被调用的情况。

准备过程

客户端首先创建了 PriceVisitor 对象，并同时创建 Pc 对象。由于 Pc 对象是一个树枝节点，因此，它的创建会造成子节点对象同时被创建。因此，IntegratedBoard 对象和 HardDisk 对象以及 Case 对象会同时被创建。系统的时序图的第一部分如下图所示。



同样，由于 IntegratedBoard 又是一个树枝节点对象，因此，它的创建会导致它所包含的下级节点对象，也就是 MainBoard 和 Cpu 对象被创建。系统时序图的第二部分如下图所示。



行动方法被调用过程的第一步

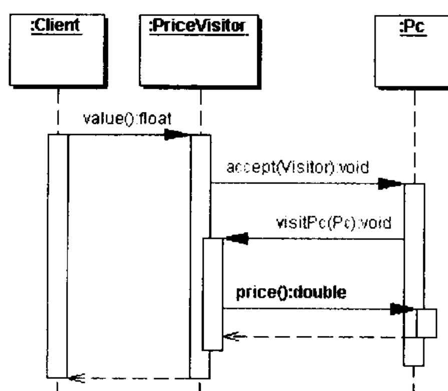
由于结构对象被省略，因此并没有结构对象的行动方法。但是，客户端可以自己承担

这个责任。同样由于时序图过长，下面将之分为几个部分讲解。

首先是客户端调用 PriceVisitor 的商业方法 value()，而 PriceVisitor 将这个调用委派给 Pc 对象。

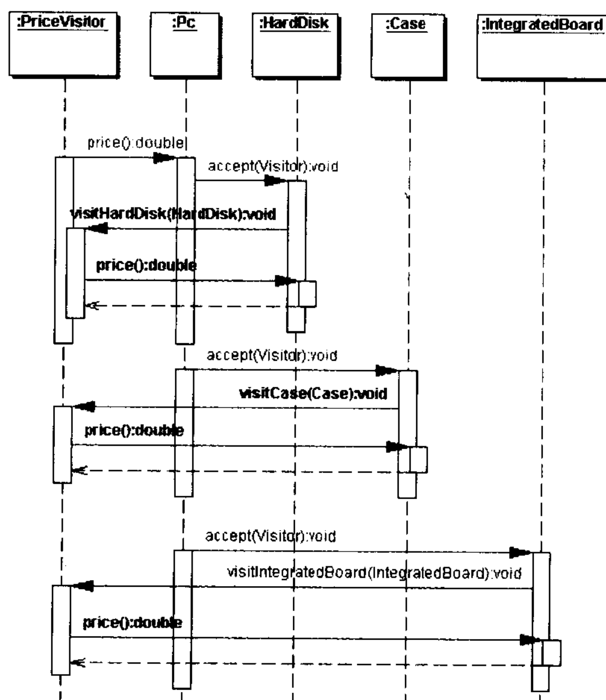
PriceVisitor 调用 Pc 对象的接受方法，而将自己作为参量传入。Pc 对象得到调用者的引用时，反过来调用 PriceVisitor 对象的 visitPc()方法，将自己的引用传入。此时 PriceVisitor 对象再次向 Pc 对象委派 price()方法，从而得到 Pc 的价格。

这是一个典型的双重分派过程，最后的结果取决于被调用方和自己的类型。系统时序图的第三部分如下图所示。



行动方法被调用过程的第二步

由于 Pc 不是一个树叶节点而是一个有内部结构，包含了 HardDisk 对象、Case 对象以及 IntegratedBoard 对象的树枝节点，因此，调用 Pc 对象的 price()方法不是一个单纯的过程，而是一个复杂的向子对象的委派过程。系统的第四部分时序图就显示了这一点，如下图所示。



从上面的图中可以看到，Pc 对象在接到 price()方法的调用后，立即调用 HardDisk 对象的接受方法，并将 PriceVisitor 对象当做参量传入。这时候，与前面的双重分派过程一模一样的过程发生了：HardDisk 对象得到调用者的引用时，反过来调用 PriceVisitor 对象的 visitHardDisk()方法，将自己的引用传入。此时，PriceVisitor 对象再次向 HardDisk 对象委派 price()方法，从而得到 HardDisk 的价格。

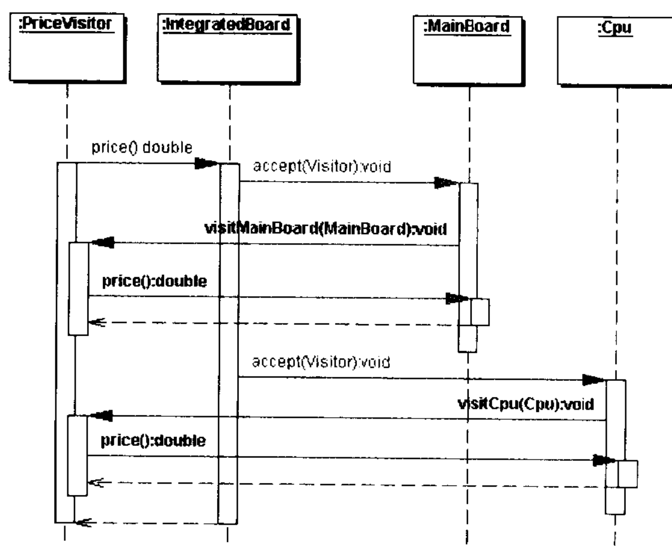
Pc 对象在接到 price()方法的调用后，同时还调用 Case 对象的接受方法，并将 PriceVisitor 对象当做参量传入。通过双重分派过程，Case 对象得到调用者的引用时，反过来调用 PriceVisitor 对象的 visitCase()方法，将自己的引用传入。此时，PriceVisitor 对象再次向 Case 对象委派 price()方法，从而得到 Case 的价格。

Pc 对象还有一个子对象，即 IntegratedBoard，因此，Pc 对象在接到 price()方法的调用后，同时还调用 IntegratedBoard 对象的接收方法，并将 PriceVisitor 对象当做参量传入。这时候，IntegratedBoard 对象得到调用者的引用时，反过来调用 PriceVisitor 对象的 visitIntegratedBoard()方法，将自己的引用传入。此时 PriceVisitor 对象再次向 IntegratedBoard 对象委派 price()方法，从而得到 IntegratedBoard 的价格。

行动方法被调用过程的第三步

IntegratedBoard 对象是一个树枝对象，具有内部结构，含有两个树叶对象：MainBoard 和 Cpu 对象。对 IntegratedBoard 对象的商业方法的调用，同样会导致向 MainBoard 和 Cpu 对象的两次双重分派过程。

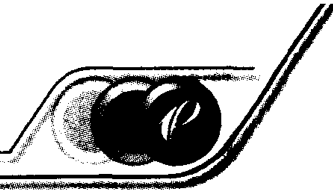
第一次双重分派导致 MainBoard 对象的价格被传到 PriceVisitor 对象；第二次双重分派导致 Cpu 对象的价格被传到 PriceVisitor 对象。系统时序图的第五部分如下图所示。



由于 Pc 对象含有六个子对象，因此，Pc 对象的价格计算导致了六次双重分派。

系统源代码

为了方便读者学习，下面给出这个系统的所有源代码。



访问者角色

首先是抽象访问者角色的源代码，如代码清单 16 所示，这个角色声明出所有的访问方法。

代码清单 16：抽象访问者角色 Visitor 的源代码

```
package com.javapatterns.visitor.inventory;
public abstract class Visitor
{
    public abstract void visitHardDisk(HardDisk e);
    public abstract void visitMainBoard(MainBoard e);
    public abstract void visitCpu(Cpu e);
    public abstract void visitPc(Pc e);
    public abstract void visitCase(Case e);
    public abstract void visitIntegratedBoard(IntegratedBoard e);
}
```

访问方法的重载

看到这些访问方法，有些读者可能会问，为什么不使用多态性原则将它们的名字重载为 visit()呢？

一般而言，Java 设计师使用方法的重载要经历如下的一个过程：

（1）初学者不用。他们还没有认识到在 Java 语言中使用重载的好处。

（2）中等程度者到处用。程序员认识到在 Java 语言中使用重载的好处，于是，凡是可以使用重载的地方一定不会忘记使用重载。比如在上面的访问者对象中会出现多个访问方法，这些访问方法显然具有多态性，可以使用重载。

（3）经过一些磨练之后，资深的设计师们认识到，原来重载在很多情况下会造成不必要的困惑，从而明白在什么情况下应当避免使用重载。仍以上面的系统为例，如果系统中出现 20 个节点，那么每个访问者对象就会有 20 个访问方法。如果使用方法重载，所有的 20 个方法都会有同样的名字。如何维护这样的代码，是一个头痛的问题。

使用重载的好处是可以强调这些方法所具有的多态性和分析的一致性。尽管它们所使用的参量不同，但是太多的同名方法会使阅读代码的人不容易看清楚不同的重载方法之间的区别。过多地使用重载会使一个不大的系统变得不必要的复杂，并导致一些很难发现的错误。

由于这里不使用重载就可以清楚地看到这些访问方法的接受者，因此是一个较好的选择。

下面是具体访问者角色 PriceVisitor 的源代码，如代码清单 17 所示。这个具体类实现了抽象访问者角色所规定的接口，并且给出了 value()商业方法。

代码清单 17：具体访问者角色 PriceVisitor 的源代码

```
package com.javapatterns.visitor.inventory;
public class PriceVisitor extends Visitor
{
    private float total;
```

```
/**
 *  构造子
 */
public PriceVisitor()
{
    total = 0;
}
/**
 *  商业方法
 */
public float value()
{
    return total;
}
/**
 *  访问方法
 */
public void visitHardDisk(HardDisk e)
{
    total += e.price();
}
/**
 *  访问方法
 */
public void visitMainBoard(MainBoard e)
{
    total += e.price();
}
/**
 *  访问方法
 */
public void visitCpu(Cpu e)
{
    total += e.price();
}
/**
 *  访问方法
 */
public void visitPc(Pc e)
{
    total += e.price();
}
/**
 *  访问方法
 */
public void visitCase(Case e)
```

```
    {
        total += e.price();
    }
    /**
     * 访问方法
     */
    public void visitIntegratedBoard(IntegratedBoard e)
    {
        total += e.price();
    }
}
```

下面是具体访问者角色 InventoryVisitor 的源代码，它实现了抽象访问者角色的接口，还给出了一个商业方法 size()，如代码清单 18 所示。

代码清单 18：具体访问者 InventoryVisitor 的源代码

```
package com.javapatterns.visitor.inventory;
import java.util.Vector;
public class InventoryVisitor extends Visitor
{
    private Vector inv;
    /**
     * 构造子
     */
    public InventoryVisitor()
    {
        inv = new Vector(10,5);
    }
    /**
     * 商业方法
     */
    public int size()
    {
        return inv.size();
    }
    /**
     * 访问方法
     */
    public void visitHardDisk(HardDisk e)
    {
        inv.addElement(e);
    }
    /**
     * 访问方法
     */
    public void visitMainBoard(MainBoard e)
    {
```

```
            inv.addElement(e);
        }
        /**
         *  访问方法
         */
        public void visitCpu(Cpu e)
        {
            inv.addElement(e);
        }
        /**
         *  访问方法
         */
        public void visitPc(Pc e)
        {
            inv.addElement(e);
        }
        /**
         *  访问方法
         */
        public void visitCase(Case e)
        {
            inv.addElement(e);
        }
        /**
         *  访问方法
         */
        public void visitIntegratedBoard(IntegratedBoard e)
        {
            inv.addElement(e);
        }
    }
```

节点对象

下面是抽象节点角色 Equipment 的源代码，如代码清单 19 所示。这个抽象类给出了接受方法 accept()和商业方法 price()的声明。

代码清单 19：抽象节点角色 Equipment 的源代码

```
package com.javapatterns.visitor.inventory;
abstract class Equipment
{
    /**
     *  接受方法
     */
    public abstract void accept(Visitor vis);
    /**
     *  商业方法
```



```
    */
    public abstract double price();
}
```

具体节点类 MainBoard 实现了抽象节点 Equipment 的接口，并且给出了接受方法 accept()和商业方法 price()的实现，如代码清单 20 所示。

代码清单 20：具体节点角色 MainBoard 的源代码

```
package com.javapatterns.visitor.inventory;
public class MainBoard extends Equipment
{
    /**
     *  商业方法
     */
    public double price()
    {
        return 100.00;
    }
    /**
     *  接受方法
     */
    public void accept(Visitor v)
    {
        System.out.println("MainBoard has been visited.");
        v.visitMainBoard(this);
    }
}
```

具体节点类 HardDisk 实现了抽象节点 Equipment 的接口，并且给出了接受方法 accept()和商业方法 price()的实现，如代码清单 21 所示。

代码清单 21：具体节点角色 HardDisk 的源代码

```
package com.javapatterns.visitor.inventory;
public class HardDisk extends Equipment
{
    /**
     *  商业方法
     */
    public double price()
    {
        return 200.00;
    }
    /**
     *  接受方法
     */
    public void accept(Visitor v)
    {
```

```
        System.out.println("HardDisk has been visited.");
        v.visitHardDisk(this);
    }
}
```

具体节点类 Cpu 实现了抽象节点 Equipment 的接口，并且给出了接受方法 accept()和商业方法 price()的实现，如代码清单 22 所示。

代码清单 22：具体节点角色 Cpu 的源代码

```
package com.javapatterns.visitor.inventory;
public class Cpu extends Equipment
{
    /**
     *  商业方法
     */
    public double price()
    {
        return 800.00;
    }
    /**
     *  接受方法
     */
    public void accept(Visitor v)
    {
        System.out.println("CPU has been visited.");
        v.visitCpu(this);
    }
}
```

具体节点类 Case 实现了抽象节点 Equipment 的接口，并且给出了接受方法 accept()和商业方法 price()的实现，如代码清单 23 所示。

代码清单 23：具体节点角色 Case 的源代码

```
package com.javapatterns.visitor.inventory;
public class Case extends Equipment
{
    /**
     *  商业方法
     */
    public double price()
    {
        return 30.00;
    }
    /**
     *  接受方法
     */
    public void accept(Visitor v)
```

```
    {
        System.out.println("Case has been visited.");
        v.visitCase(this);
    }
}
```

复合节点

抽象复合节点类实现了抽象节点 Equipment 的接口，并且给出了接受方法 accept()和商业方法 price()的实现，如代码清单 24 所示。注意这个类持有一个对所有子节点的聚集，提供了一个相应的聚集管理方法。它的接受方法需要向所有的子节点委派，它的商业方法也同样向所有的子节点委派。

代码清单 24：抽象复合节点 Composite 的源代码

```
package com.javapatterns.visitor.inventory;
abstract class Composite extends Equipment
{
    private Vector parts = new Vector(10);
   /**
    *  构造子
    */
    public Composite()
    {
    }
   /**
    *  商业方法
    */
    public double price()
    {
        double total=0;
        for (int i=0; i < parts.size(); i++)
        {
            total += ((Equipment) parts.get(i)).price();
        }
        return total;
    }
   /**
    *  接受方法
    */
    public void accept(Visitor v)
    {
        for (int i=0; i < parts.size(); i++)
        {
            ((Equipment) parts.get(i)).accept(v);
        }
    }
}
```

作为 Composite 的子类，具体类 IntegratedBoard 类扩展了抽象类 Composite，它一方面改善了接受方法，另一方面在自己的构造子里面指明了下层对象，如代码清单 25 所示。

代码清单 25：具体复合节点角色 IntegratedBoard 的源代码

```
package com.javapatterns.visitor.inventory;
public class IntegratedBoard extends Composite
{
    /**
     *  构造子
     */
     public IntegratedBoard()
     {
         super.add(new MainBoard());
         super.add(new Cpu());
     }
    /**
     *  接受方法
     */
     public void accept(Visitor v)
     {
         System.out.println("IntegratedBoard has been visited.");
         super.accept(v);
     }
}
```

Pc 同样也是 Composite 的子类，它扩展了抽象类 Composite，一方面改善了接受方法，另一方面在自己的构造子里面指明了下层对象，如代码清单 26 所示。



它的下层对象之一是 IntegratedBoard 对象，它本身就是树枝对象。

代码清单 26：具体复合节点 Pc 的源代码

```
package com.javapatterns.visitor.inventory;
public class Pc extends Composite
{
    /**
     *  构造子
     */
     public Pc()
     {
         super.add(new IntegratedBoard());
         super.add(new HardDisk());
         super.add(new Case());
     }
    /**
```

```
    *  接受方法
    */
    public void accept(Visitor v)
    {
        System.out.println("Pc has been visited.");
        super.accept(v);
    }
}
```

客户端

下面是一个示意性的客户端对象的源代码，如代码清单 27 所示。它调用了 PriceVisitor 对象以取得零件的总价格，而且调用了 InventorVisitor 以取得所有零件的总数目。

代码清单 27：示意性的客户端的源代码

```
package com.javapatterns.visitor.inventory;
public class Client
{
    private static PriceVisitor pv;
    private static InventoryVisitor iv;
    private static Equipment equip;
    public static void main(String[] argv)
    {
        equip = new Pc();
        pv = new PriceVisitor();
        equip.accept(pv);
        System.out.println("Price: " + pv.value() );
        System.out.println("\n");
        iv = new InventoryVisitor();
        equip.accept(iv);
        System.out.println("Number of parts: " + iv.size());
    }
}
```

系统运行的结果

客户端运行时会将结果打印到屏幕上，如代码清单 28 所示。

代码清单 28：示意性的客户端源代码

```
Pc has been visited.
IntegratedBoard has been visited.
MainBoard has been visited.
CPU has been visited.
HardDisk has been visited.
Case has been visited.
Price: 1130.0
Pc has been visited.
```

```
IntegratedBoard has been visited.
MainBoard has been visited.
CPU has been visited.
HardDisk has been visited.
Case has been visited.
Number of parts: 4
```

可以看出，这个示意性的客户端的运行结果分成两大部分，第一部分是对零件总价格的统计；第二部分是对零件总数目的统计。

访问者对象的状态积累

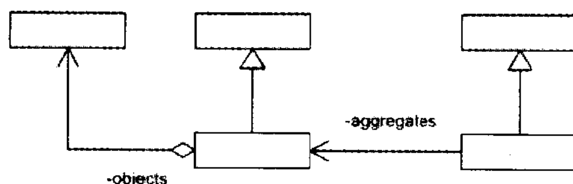
在这个系统中，PriceVisitor 和 InventorVisitor 的用意就是在访问过程中积累状态。前者积累的状态是零件的总价格；后者积累的状态是零件的总数目。

53.9 与访问者模式有关的模式

迭代模式

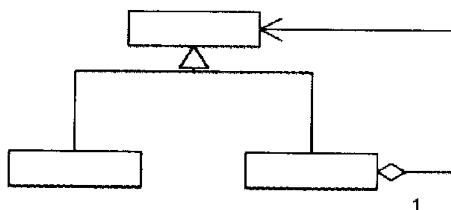
如果所浏览的结构对象是线性的，使用迭代模式而不是访问者模式也是可以的。

在使用迭代模式遍历一个树结构时，可以调用访问者模式。迭代模式的简略类图如下图所示。



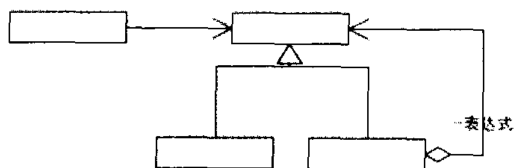
合成模式

访问者模式常常浏览符合合成模式的一些结构对象。合成模式的简略类图如下图所示。



解释器模式

访问者模式可以参与解释器模式的实现。解释器模式的简略类图如下图所示。



参考文献

[NORDBERG98] Martin E. Nordberg III. Default and Extrinsic Visitor, in Pattern Languages of Program Design 3. edited by Robert Martin, Dirk Riehle, Frank Buschmann, published by Addison Wesley, 1998

[BLOSSER98] Jeremy Blosser. Relfect ont the Visitor design pattern. JavaWorld, 1998

第 54 章　解释器（Interpreter）模式

解释器模式是类的行为模式[GOF95]。给定一个语言之后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。

54.1　引　　言

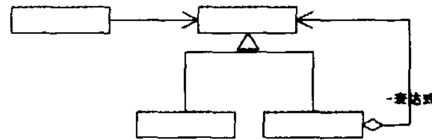
问题

如果某一类问题一再地发生的话，那么一个有意义的做法就是将此类问题的各个实例表达为一个简单语言中的语句。这样就可以建造一个解释器，通过解释这些语句达到解决问题的目的。

例如，依照一个匹配模式搜寻字符串便是一个常见的问题。与其为每一个匹配模式建造一个特定的算法，不如建造一个一般性的算法处理各种常规表达式。当接到一个指定的常规表达式时，系统使用一个解释器解释这个常规表达式，从而对字符串进行匹配。

再比如 VBA（Visual Basic for Applications）不仅仅出现在微软的 Office 系列软件中，并且可以供第三厂家出产的软件嵌入使用；Crystal Reports 报表生成软件也包括了一个便于使用的宏语言，使用户可以执行较为复杂的命令操作。一般而言，将 VBA 或者其他的语言软件嵌入到自己的软件产品中，可以使产品定制化（Customization）能力大大增强，但是这些宏语言引擎往往都很昂贵。

现在要介绍的解释器模式将描述怎样在有了一个简单的文法后，使用模式设计解释这些语句。熟悉了这个模式以后，一个没有接收过形式语言和编译器的正规训练的设计师也可以自行设计一个简单的解释器，以便为客户端提供一个简单语言，或者在系统内部使用一个简单语言描述一个合适的问题。解释器模式的简略类图如右图所示。



语言、解释器和浏览器

解释器模式只描述解释器是怎样工作的，并不指明怎样在运行时创建新的解释器。虽然广义地讲，解释器不一定要有一个浏览器（Parser），但是使用浏览器仍然是最常见的建立解释器的办法。一个浏览器可以从一个文件或命令行读入文字性命令，并创建解释器。

浏览器是一种能够识别文字并将文字按照一定规则进行分解以便进一步处理的对象。

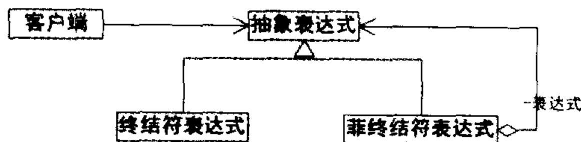
浏览器能够识别的字符串叫做语言。通常建立的小型计算机语言是与环境无关的语言，也就是遵循一定的文法的文字模式。所谓文法，便是决定怎样将语言的元素组合起来的规则的集合。浏览器便是根据组合规则将字符串分解的。

抽象地讲，语言并不一定是以字符串的形式表达的。在解释器模式里面所提到的语言是指任何解释器对象能够解释的任何组合。在解释器模式中，需要定义一个代表文法的命令类的等级结构，也就是一系列的组合规则；每一个命令对象都有一个解释方法，代表对命令对象的解释。

命令对象的等级结构中的对象的任何排列组合都是一个语言，而浏览器的工作便是将一个文字性语言翻译成为等效的解释器语言。因此，解释器往往需要浏览器。由于篇幅和选题的限制，本书不能讲述浏览器的创建方法，但是读者可以参考“专题：观察者模式与SAX2 浏览器”一章中关于 SAX 浏览器的描述，以及有关的书籍。

54.2 解释器模式的结构

下面就以一个示意性的系统为例，讨论解释器模式的结构。系统的结构图如下图所示。



模式所涉及的角色如下所示。

（1）抽象表达式（Expression）角色：声明一个所有的具体表达式角色都需要实现的抽象接口。这个接口主要是一个 interpret()方法，称做解释操作。

（2）终结符表达式（Terminal Expression）角色：这是一个具体角色。

- 实现了抽象表达式角色所要求的接口，主要是一个 interpret()方法；
- 文法中的每一个终结符都有一个具体终结表达式与之相对应。

（3）非终结符表达式（Nonterminal Expression）角色：这是一个具体角色。

- 文法中的每一条规则 $R=R_1R_2...R_n$ 都需要一个具体的非终结符表达式类；
- 对每一个 R1R2...R*n* 中的符号都持有一个静态类型为 Expression 的实例变量；
- 实现解释操作，即 interpret()方法。解释操作以递归方式调用上面所提到的代表 R1R2...R*n* 中的各个符号的实例变量。

（4）客户端（Client）角色：代表模式的客户端它有以下功能。

- 建造一个抽象语法树（AST 或者 Abstract Syntax Tree）；
- 调用解释操作 interpret()。

在一般情况下，模式还需要一个环境角色。

（5）环境（Context）角色：提供解释器之外的一些全局信息，比如变量的真实量值等。

54.3 一个示意性的实现

问题

为了说明解释器模式的实现办法，这里给出一个最简单的文法和对应的解释器模式的实现，这就是模拟 Java 语言中对布尔表达式进行操作和求值。这个例子是 GOF 著作中的经典例子，熟悉 C 语言的读者可以参考原书的描述，熟悉 Java 语言的读者可以阅读本书在下面所给出的 Java 解答。

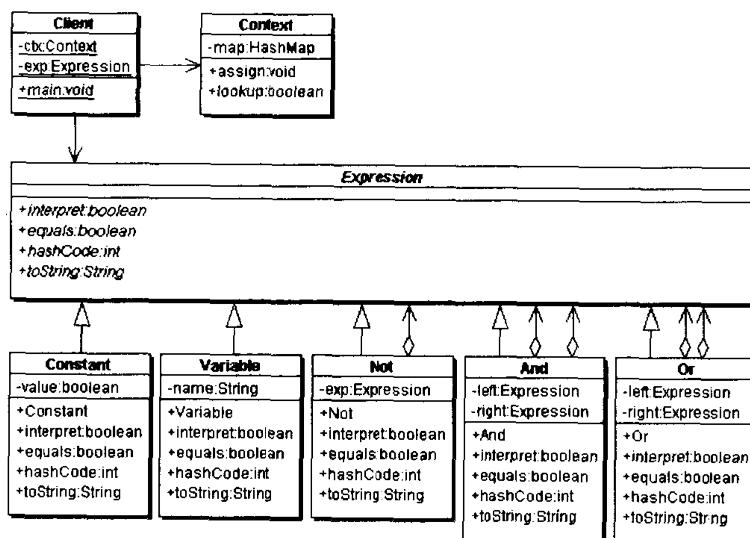
在这个语言中终结符是布尔变量，也就是常量 true 和 false。非终结符表达式包含运算符 and，or 和 not 等布尔表达式。这个简单的文法如下（为简单起见，省略了操作符的优先次序和括号的运用）：

```
Expression   ::= Constant | Variable | Or | And | Not
And          ::= Expression 'AND' Expression
Or           ::= Expression 'OR' Expression
Not          ::= 'NOT' Expression
Variable     ::= 任何标识符
Constant     ::= 'true' | 'false'
```

抽象语法树（AST）的每一个节点都代表一个语句，而在每一个节点上都可以执行解释方法。这个解释方法的执行就代表这个语句被解释。由于每一个语句都代表一个常见的问题的实例，因此每一个节点上的解释操作都代表对一个问题实例的解答。

设计

解释器模式系统的结构图如下图所示。

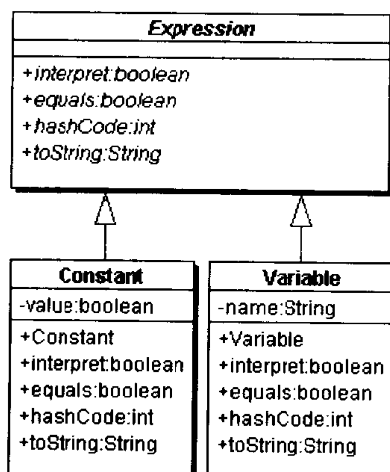


角色

Expression
+interpret:boolean
+equals:boolean
+hashCode:int
+toString:String

首先，抽象表达式角色声明了由三个方法组成的接口，其中最重要的就是解释操作 interpret()方法，抽象表达式角色的结构图如右图所示。

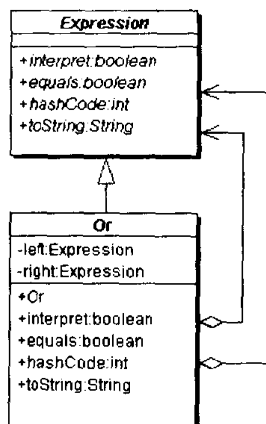
其次是终结表达式角色，如下图所示。



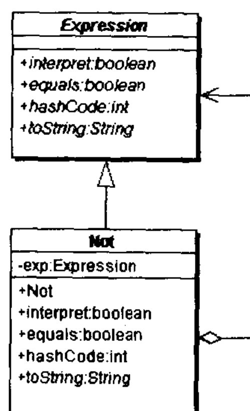
再次是非终结表达式角色，非终结表达式类 And 的结构图如下图所示。

Expression
+interpret:boolean
+equals:boolean
+hashCode:int
+toString:String
And
-left:Expression
-right:Expression
+And
+interpret:boolean
+equals:boolean
+hashCode:int
+toString:String

上面的 And 非终结表达式代表一个二元运算，运算的文法是 R=R1R2，或者 Expression 'And' Expression，因此，它会有两个合成关系指向 Expression 类。非终结表达式类 Or 的结构图如下图所示。



与前面的 And 相似，非终结表达式 Or 也是一个二元运算，运算的文法是 R=R1R2，或者 Expression 'Or' Expression，因此，它也有两个合成关系指向 Expression 类。非终结表达式类 Not 的结构图如下图所示。



非终结表达式 Not 代表一个一元运算，文法 R=R1，或者 'Not' Expression，因此它只有一个合成关系指向 Expression 类。

源代码

这个实现的源代码如下所示。首先是抽象表达式角色的源代码，如代码清单 1 所示。

代码清单 1：抽象角色 Expression 的源代码

```
package com.javapatterns.interpreter;
/**
 * 这个抽象类代表终结类和非终结类的抽象化
 * 其中终结类和非终结类来自下面的文法
 * Expression ::=
 *              Expression AND Expression
 *            | Expression OR Expression
 *            | NOT Expression
 *            | Variable
 *            | Constant
```




```
 * Variable ::= ... // 可以打印出的非空白字符串
 * Contant   ::= "true" | "false"
 */
public abstract class Expression
{
    /**
     * 以环境类为准，本方法解释给定的任何一个表达式
     */
    public abstract boolean interpret(Context ctx);
    /**
     * 检验两个表达式在结构上是否相同
     */
    public abstract boolean equals(Object o);
    /**
     * 返回表达式的 hash code
     */
    public abstract int hashCode();
    /**
     * 将表达式转换成字符串
     */
    public abstract String toString();
}
```

这个抽象角色规定出所有的具体表达式角色必须实现的接口，在这里就是 interpret()、equals()、hasCode()以及 toString()等方法。一个 Constant 对象代表一个布尔常量，如代码清单 2 所示。

代码清单 2：Constant 类的源代码

```
package com.javapatterns.interpreter;
public class Constant extends Expression
{
    private boolean value;
    // 构造子
    public Constant(boolean value)
    {
        this.value = value;
    }
    /**
     * 解释操作
     */
    public boolean interpret(Context ctx)
    {
        return value;
    }
    /**
     * 检验两个表达式在结构上是否相同
```



```
    */
    public boolean equals(Object o)
    {
        if (o != null && o instanceof Constant)
        {
            return this.value == ((Constant) o).value;
        }
        return false;
    }
    /**
     * 返回表达式的 hash code
     */
    public int hashCode()
    {
        return (this.toString()).hashCode();
    }
    /**
     * 将表达式转换成字符串
     */
    public String toString()
    {
        return new Boolean(value).toString();
    }
}
```

在调用时，通过向构造子传入一个布尔常量，得到一个包装了这个布尔常量的对象：new Constant(true)。一个 Variable 对象代表一个有名变量，如代码清单 3 所示。

代码清单 3：Variable 类的源代码

```
package com.javapatterns.interpreter;
public class Variable extends Expression
{
    private String name;
    /**
     * 构造子
     */
    public Variable(String name)
    {
        this.name = name;
    }
    /**
     * 解释操作
     */
    public boolean interpret(Context ctx)
    {
        return ctx.lookup(this);
    }
```

```
        /**
         * 检验两个表达式在结构上是否相同
         */
        public boolean equals(Object o)
        {
            if (o != null && o instanceof Variable)
            {
                return this.name.equals(
                    ((Variable) o).name);
            }
            return false;
        }
        /**
         * 返回表达式的 hash code
         */
        public int hashCode()
        {
            return (this.toString()).hashCode();
        }
        /**
         * 将表达式转换成字符串
         */
        public String toString()
        {
            return name;
        }
    }
```

在使用 Variable 类时，需要将变量名传入构造子中，例如：new Variable("x")，就声明了一个名字为 x 的变量。

下面是代表逻辑与操作的 And 类，表示由两个布尔表达式通过逻辑与操作给出一个新的布尔表达式的操作，如代码清单 4 所示。

代码清单 4：And 类的源代码

```
    package com.javapatterns.interpreter;
    public class And extends Expression
    {
        private Expression left, right;
        /**
         * 构造子
         */
        public And(Expression left,
            Expression right)
        {
            this.left = left;
            this.right = right;
```

```
    }
    /**
     * 解释操作
     */
    public boolean interpret(Context ctx)
    {
        return left.interpret(ctx) &&
            right.interpret(ctx);
    }
    /**
     * 检验两个表达式在结构上是否相同
     */
    public boolean equals(Object o)
    {
        if (o != null && o instanceof And)
        {
          return this.left.equals(((And) o).left) &&
              this.right.equals(((And) o).right);
        }
        return false;
    }
    /**
     * 返回表达式的 hash code
     */
    public int hashCode()
    {
        return (this.toString()).hashCode();
    }
    /**
     * 将表达式转换成字符串
     */
    public String toString()
    {
        return "(" + left.toString()
            + " AND "
            + right.toString() + ")";
    }
}
```

在使用时，new And(x, y)代表 x And y。

下面是代表逻辑或操作的 Or 类，代表由两个布尔表达式通过逻辑或操作给出一个新的布尔表达式的操作，如代码清单 5 所示。

代码清单 5：Or 类的源代码

```
package com.javapatterns.interpreter;
public class Or extends Expression
```




```
{
    private Expression left, right;
    /**
     * 构造子
     */
    public Or(Expression left,
        Expression right)
    {
        this.left = left;
        this.right = right;
    }
    /**
     * 解释操作
     */
    public boolean interpret(Context ctx)
    {
        return left.interpret(ctx)
            || right.interpret(ctx);
    }
    /**
     * 检验两个表达式在结构上是否相同
     */
    public boolean equals(Object o)
    {
        if (o != null && o instanceof Or)
        {
            return this.left.equals(((Or) o).left) &&
                this.right.equals(((Or) o).right);
        }
        return false;
    }
    /**
     * 返回表达式的 hash code
     */
    public int hashCode()
    {
        return (this.toString()).hashCode();
    }
    /**
     * 将表达式转换成字符串
     */
    public String toString()
    {
        return "(" + left.toString()
            + " OR "
            + right.toString() + ")";
```

```
    }
}
```

在使用时，new Or(x, y)代表 x Or y。

下面是代表逻辑非操作的 Not 类，代表由一个布尔表达式通过逻辑非操作给出一个新的布尔表达式的操作，如代码清单 6 所示。

代码清单 6：Not 类的源代码

```
package com.javapatterns.interpreter;
public class Not extends Expression
{
    private Expression exp;
    /**
     * 构造子
     */
    public Not(Expression exp)
    {
        this.exp = exp;
    }
    /**
     * 解释操作
     */
    public boolean interpret(Context ctx)
    {
        return !exp.interpret(ctx);
    }
    /**
     * 检验两个表达式在结构上是否相同
     */
    public boolean equals(Object o)
    {
        if (o != null && o instanceof Not)
        {
            return this.exp.equals(
                ((Not) o).exp);
        }
        return false;
    }
    /**
     * 返回表达式的 hash code
     */
    public int hashCode()
    {
        return (this.toString()).hashCode();
    }
    /**
```

```
     * 将表达式转换成字符串
     */
    public String toString()
    {
        return " (Not " + exp.toString() + ")";
    }
}
```

在使用时，new Not(x)代表 Not x。

环境类定义出从变量到布尔值（也就是 true 和 false）的一个映射，如代码清单 7 所示。

代码清单 7：Context 类的源代码

```
package com.javapatterns.interpreter;
import java.util.HashMap;
public class Context
{
    private HashMap map = new HashMap();
    public void assign(Variable var,
        boolean value)
    {
        map.put(var, new Boolean(value));
    }
    public boolean lookup(Variable var)
        throws IllegalArgumentException
    {
        Boolean value = (Boolean) map.get(var);
        if (value == null)
        {
            throw new IllegalArgumentException();
        }
        return value.booleanValue();
    }
}
```

下面是一个示意性的客户端类的实现，如代码清单 8 所示。

代码清单 8：Client 类的源代码

```
package com.javapatterns.interpreter;
public class Client
{
    private static Context ctx;
    private static Expression exp ;
    public static void main(String[] args)
    {
        ctx = new Context();
        Variable x = new Variable("x");
```

```
        Variable y = new Variable("y");
        Constant c = new Constant(true);
        ctx.assign(x, false);
        ctx.assign(y, true);
        exp = new Or( new And(c, x) ,
            new And(y, new Not(x)));
        System.out.println( "x = " + x.interpret(ctx));
        System.out.println( "y = " + y.interpret(ctx));
        System.out.println( exp.toString()
            + " = " + exp.interpret(ctx));
    }
}
```

下面的客户端定义了一个布尔表达式，如代码清单 9 所示。

代码清单 9：Client 类的源代码

```
(true And x) Or ( y And (Not x))
```

在运行时会打印出如下的结果，如代码清单 10 所示。

代码清单 10：运行的结果

```
x = false
y = true
((true AND x) OR (y AND  (Not x))) = true
```

这就是解释器对这个布尔表达式解释的结果。

解释器模式适用于以下的情况：

（1）系统有一个简单的语言可供解释。

（2）一些重复发生的问题可以用这种简单的语言表达。

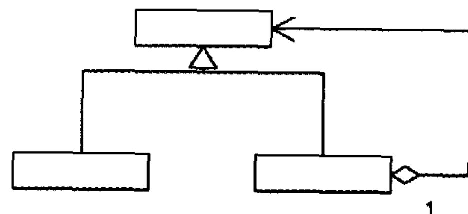
（3）效率不是主要的考虑。

54.4　有 关 模 式

解释器模式与以下的模式有关系。

合成模式

被解释的表达式所在的抽象语法树（AST）就是建立在合成模式的基础之上的。终结表达式就是树叶节点，非终结表达式就是复合（树枝）节点。合成模式的简略类图如右图所示。

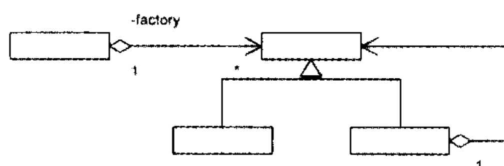


所有使用了合成模式的系统都会在一个树结构上执行某种操作，但是这并不一定构成解释器

模式。解释器模式要求这个树结构描述的是一个语言的文法，也就是说，一定要在有一个语言时，才谈得上解释器模式。

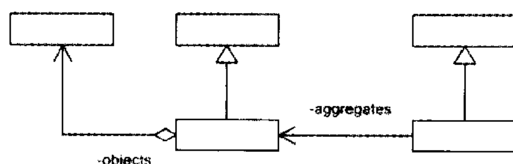
不变模式和享元模式

为了减少重复和相似的表达式对象的数量，可以将一些表达式对象设计成不变模式，从而达到使用享元模式共享这些表达式对象的目的。享元模式的简略类图如下图所示。



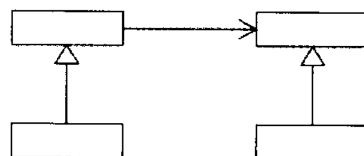
迭代子模式

在解释器模式遍历抽象语法树（AST）及其节点时，要用到迭代子模式。迭代子模式的简略类图如下图所示。



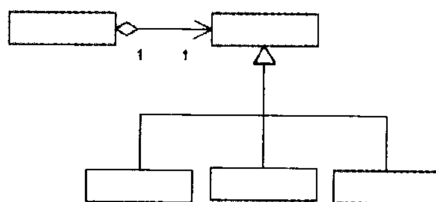
访问者模式

如果使用访问者模式将各个抽象语法树（AST）的节点中的操作转移到一个访问者类中的话，解释器模式可以有更多的灵活性。访问者模式的简略类图如下图所示。



状态模式

解释器模式中的环境对象（Context）往往可以使用状态模式实现。状态模式的简略类图如下图所示。



第 55 章　调停者（Mediator）模式

调停者模式是对象的行为模式[GOF95]。调停者模式包装了一系列对象相互作用的方式，使得这些对象不必互相明显引用。从而使它们可以较松散地耦合。当这些对象中的某些对象之间的相互作用发生改变时，不会立即影响到其他的一些对象之间的相互作用。从而保证这些相互作用可以彼此独立地变化。

请读者在阅读本章之前，首先阅读本书的“迪米特法则（LoD）”一章，并且可以在阅读过程当中与本书的“观察者（Observer）模式”一章相比较。

55.1　引　　言

不要滥用模式

从理论上讲，面向对象的编程鼓励代码的复用，而设计模式本身是经过时间检验的设计方案，因此，应当说应用设计模式便是对成功的设计方案的复用。通过设计方案的复用，可以带动代码的复用，达到提高代码复用率的作用。但是所有的理论在应用到实践中的时候，都必须对具体问题做具体分析。

随着设计模式越来越普及，有一种倾向也变得越来越明显，这就是没有经验的设计师对设计模式的盲目崇拜和过分的追求。这些设计师不是全力以赴地为他们所面临的问题找出最好的设计，而是将力气放在如何尽可能多和频繁地使用著名的模式。他们错误地认为，只要使用了这些设计模式，就可以保证一个设计方案是好的设计方案。因此，使用的模式越多，设计就越好，这就导致了很多根本没有意义的设计。在这些设计里充斥着著名的设计模式，但是设计却和系统的需要严重脱节。

要想恰到好处地在一个系统里面使用设计模式，必须做到以下几点：

（1）完全了解面临的问题，这就是说要完全了解具体情况。如果不完全了解所面临的问题，怎么能谈得上解决问题呢？

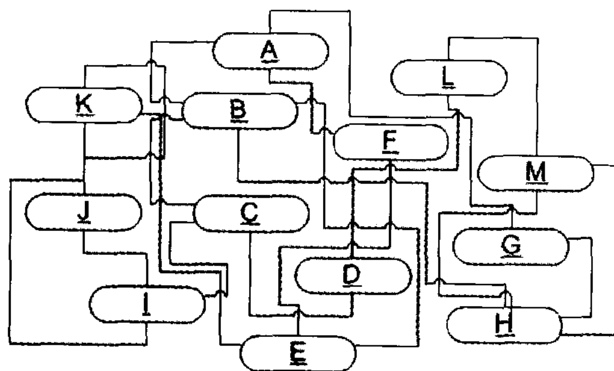
（2）完全了解模式，这就是说要十分懂得理论。如果不完全懂得所使用的理论，怎么能够正确地应用这一理论呢？

（3）非常了解怎样使用设计模式解决实际的问题，这就是说要将模式理论与具体系统需求情况相结合。如果设计师不知道一个设计模式怎样对系统设计有帮助的话，最好不要使用这个模式。不要只是因为想在简历上写上设计模式方面的经验就盲目地使用模式。

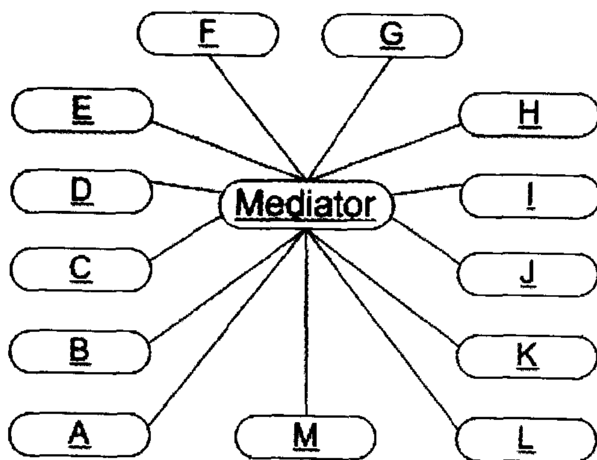
调停者模式就是一个很容易被滥用的模式。

为什么需要调停者模式

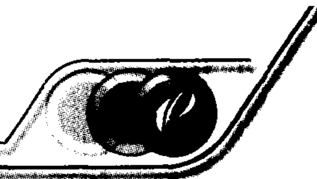
如下图所示，这个示意图中有大量的对象，这些对象既会影响别的对象，又会被别的对象所影响，因此常常叫做同事（Colleague）对象。这些同事对象通过彼此的相互作用形成系统的行为。从图中可以看出，几乎每一个对象都需要与其他的对象发生相互作用，而这种相互作用表现为一个对象与另一个对象的直接耦合。这是一个过度耦合的系统。



通过引入调停者对象（Mediator），可以将系统的网状结构变成以中介者为中心的星形结构，如下图所示。在这个星形结构中，同事对象不再通过直接的联系与另一个对象发生相互作用；相反地，它通过调停者对象与另一个对象发生相互作用。调停者对象的存在保证了对象结构上的稳定，也就是说，系统的结构不会因为新对象的引入造成大量的修改工作。



比较传统的设计方法是，面向对象的技术可以更好地协助设计师管理更为复杂的系统。一个好的面向对象的设计可以使对象之间增加协作性（Collaboration），减少耦合度（Coupling）。一个深思熟虑的设计会把一个系统分解为一群相互协作的同事对象，然后给每一个同事对象以独特的责任，恰当的配置它们之间的协作关系，使它们可以在一起工作。



对行为的封装

一个对象所封装的是对象的行为，而面向对象的设计的中心问题是如何体现系统的行为。因为一个系统的行为不仅仅存在于一个对象里面，而是同时存在于很多对象里面，并且存在于这些对象之间的相互作用之中。因此，一个设计恰当地体现系统的行为并不是一个容易的问题。

一个对象往往要与几个对象发生相互作用，就好比在一个小组中工作的人一样，每一个人都要与他的同事发生相互作用。这时，一个直截了当的、不好的设计会把这些与其他的对象直接固定到每一个对象上面。这样做的结果自然是使这些对象独立演化的难度增加，使每一个对象的变化都会影响到所有与之发生相互作用的对象。这就好比每一个在小组中工作的人都要成为其他人的经理，不仅要管理自己，还要管理所有的同事。

在现实世界中，没有一个小组能在这样过度耦合的组织结构中完成任何事情。想像一个小组只有三个人，那么三个人之间会有三个相互关系；而如果小组的规模扩大到五个人时，这种相互作用关系就增加到十个，增长的速度是组合数式的，增长速度比指数增长的速度还快。

在实际工作中，一个有效的管理制度是给这个小组配备一个小组长，负责协调小组成员之间的关系，小组的同事通过小组长可以形成整个小组的行为。这实际上就是调停者模式的精神。

什么是调停者模式

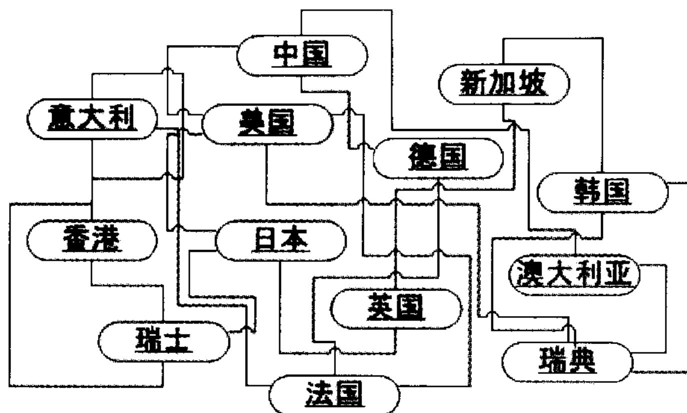
在调停者模式中，所有的成员对象都可以协调工作，但是又不直接相互管理。这些对象都与一个处于中心地位的调停者对象发生紧密的关系，由这个调停者对象进行协调工作。这个协调者对象叫做调停者（Mediator），而调停者所协调的成员对象称做同事（Colleague）对象。

在 Colleague 对象内部发生的事件会影响到所有的同事，但是这种影响不是以直接管理的方式直接传到其他的对象上的。记住在小组的成员增加时，这样的相互作用关系是以比指数更快的方式增加的。相反，这种影响仅仅直接影响到调停者对象，而调停者对象反过来会协调其他的同事，形成整个系统的行为。

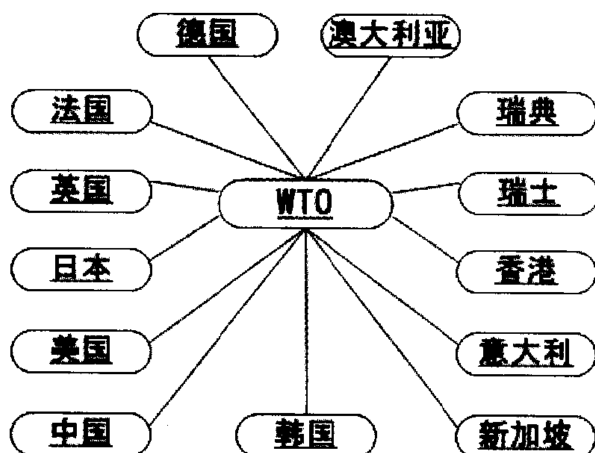
如果小组的成员增加时，调停者对象可能会面临修改，而其他的同事则可以装做不知道这个新的成员一样，不必修改。反过来，如果小组的成员之一从系统中被删除的话，调停者对象需要对此做出修改，而小组中其他的同事则不必改动。

中国加入 WTO

WTO 是一个协调组织，各个贸易地区可以经由 WTO 组织进行协调。WTO 扮演的正是调停者模式中的调停者角色，它取代了原本有各个贸易地区自行进行的相互协调和谈判的强耦合关系，各个贸易地区在加入 WTO 组织之前的贸易状态，如下图所示。



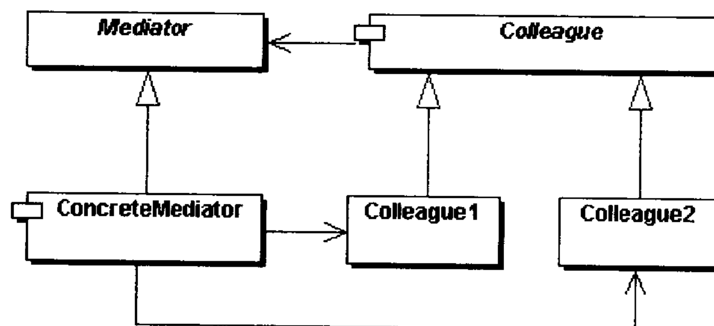
WTO 使得各个贸易地区之间的强关联转换成为较松散的耦合关系，各贸易地区加入 WTO 之后的贸易关系，如下图所示。



记住这个比喻，读者可以对调停者模式有更好的理解。

55.2　调停者模式的结构

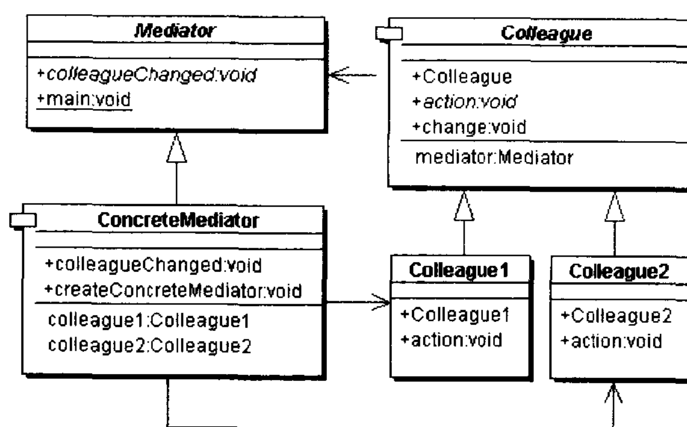
调停者模式的示意性类图如下图所示。



调停者模式包括以下几种角色：

- 抽象调停者角色：定义出同事对象到调停者对象的接口，其中主要的方法是一个（或者多个）事件方法。在有些情况下，这个抽象对象可以省略。一般而言，这个角色由一个 Java 抽象类或者 Java 对象实现。
- 具体调停者角色：从抽象调停者继承而来，实现了抽象超类所声明的事件方法。具体调停者知晓所有的具体同事类，它从具体同事对象接收消息、向具体同事对象发出命令。一般而言，这个角色由一个具体 Java 类实现。
- 抽象同事类角色：定义出调停者到同事对象的接口。同事对象只知道调停者而不知道其余的同事对象。一般而言，这个角色由一个 Java 抽象类或者 Java 对象实现。
- 具体同事类角色：所有的具体同事类均从抽象同事类继承而来。每一个具体同事类都很清楚它自己在小范围内的行为，而不知道它在大范围内的目的。在示意性的类图中，具体同事类是 Colleague1 和 Colleague2。一般而言，这个角色由一个具体 Java 类实现。

为了方便读者学习这个模式的细节，本章提供了调停者模式的一个简化的示意性实现，如下图所示。



在这个实现里，有两个具体同事类，分别为 Colleague1 和 Colleague2 以及一个具体调停者角色。

下面就是抽象同事角色的源代码，如代码清单 1 所示。

代码清单 1：抽象同事类的源代码

```
package com.javapatterns.mediator;
public abstract class Colleague
{
    private Mediator mediator;

   /**
    *  构造子，作为参量接收调停者对象
    */
    public Colleague(Mediator m)
```

```
    {
        mediator = m;
    }
    /**
     * 取值方法，提供调停者对象
     */
    public Mediator getMediator()
    {
        return mediator;
    }
    /**
     * 行动方法，由子类实现
     */
    public abstract void action();
    /**
     * 示意性的商业方法，调用此方法
     * 可以改变对象的内部状态
     */
    public void change()
    {
        mediator.colleagueChanged(this);
    }
}
```

从上面的源代码可以看出，这个抽象 Java 类声明了一个抽象的商业方法 action()，这个方法常常叫做行动方法。一个同事对象在得知其他对象有变化时，会执行这个操作。

下面是具体同事角色的源代码，如代码清单 2 所示。首先，这个具体角色实现了抽象同事类所声明的抽象行动方法 action()。在一个真实的系统中，这个方法会与商业逻辑有关。其次，这个具体类的构造子接收一个调停者类型的参量。这就是说，一个同事对象在创建之时就已经知道了调停者，正如前面所说的，每一个同事对象都仅知道调停者，而不知道其他的同事对象。

代码清单 2：具体同事类的源代码

```
package com.javapatterns.mediator;
public class Colleague1 extends Colleague
{
    /**
     * 构造子，作为参量接收调停者对象
     */
    public Colleague1(Mediator m)
    {
        super( m );
    }
    /**
     * 行动方法的具体实现
     */
```

```
    public void action()
    {
        System.out.println("This is an action from Colleague 1");
    }
}
```

下面是第二个同事对象的源代码，如代码清单 3 所示。

代码清单 3：具体同事类的源代码

```
package com.javapatterns.mediator;
public class Colleague2 extends Colleague
{
    /**
     *  构造子，作为参量接收调停者对象
     */
    public Colleague2(Mediator m)
    {
        super( m );
    }
    /**
     *  行动方法的具体实现
     */
    public void action()
    {
        System.out.println("This is an action from Colleague 2");
    }
}
```

下面是抽象调停者角色的源代码，如代码清单 4 所示。

代码清单 4：抽象调停者类的源代码

```
package com.javapatterns.mediator;
abstract class Mediator
{
    /**
     *  事件方法，由子类实现
     */
    public abstract void colleagueChanged(Colleague c);

    public static void main(String args[])
    {
        ConcreteMediator mediator = new ConcreteMediator();
        mediator.createConcreteMediator();
        Colleague c1 = new Colleague1(mediator);
        Colleague c2 = new Colleague2(mediator);
        mediator.colleagueChanged(c1);
    }
```

```
}
```

可以看出，上面这个抽象类声明了一个抽象方法 colleagueChanged()，这就是所谓的事件方法。当自身的状态发生了变化的时候，一个同事对象可以调用这个事件方法来通知调停者，从而更新所有有关的同事对象。

下面是具体调停者角色的源代码，如代码清单 5 所示。可以看出，这个类提供了抽象调停者角色所声明的事件方法 colleagueChanged()的具体实现。显然，当这个方法被调用时，会通知所有有关的同事对象更新自己的状态。

此外，具体调停者角色提供了 createConcreteMediator()方法，以便在需要的时候将所有的同事类实例化，并通过适当的取值方法提供这些同事对象。可以看出，这是一个工厂方法。它并不自己提供所创建的产品对象，而是通过其他的方法提供这些对象。在这里提供新的产品实例的方法就是 getColleague1()和 getColleague2() 等。

代码清单 5：具体调停者类的源代码

```
package com.javapatterns.mediator;
public class ConcreteMediator extends Mediator
{
    private Colleague1 colleague1;
    private Colleague2 colleague2;
   /**
    *  事件方法的具体实现
    */
    public void colleagueChanged( Colleague c )
    {
        colleague1.action();
        colleague2.action();
    }

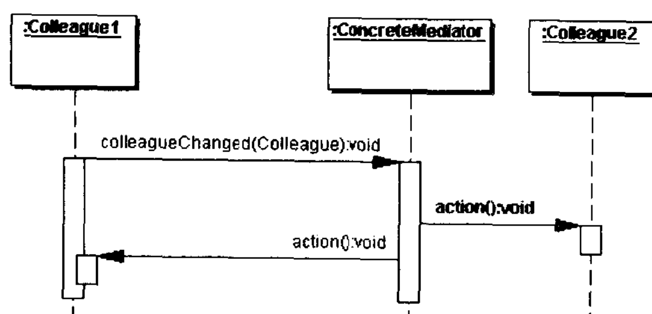
   /**
    *  工厂方法，创建同事对象
    */
    public void createConcreteMediator()
    {
        colleague1 = new Colleague1(this);
        colleague2 = new Colleague2(this);
    }

   /**
    *  取值方法，提供同事对象
    */
    public Colleague1 getColleague1()
    {
        return colleague1 ;
    }
```

```
    /**
     *   取值方法，提供同事对象
     */
    public Colleague2 getColleague2()
    {
        return colleague2 ;
    }
}
```

调停者模式的时序

在某一个同事对象的状态发生变化时，这个同事对象需要通过调停者对象的事件方法通知其他的同事对象采取行动。比如 Colleague1 对象发生了状态变化，这时它调用 Mediator 对象的事件方法 colleagueChanged()，并将自己作为参量传给 Mediator 对象；后者则根据商业逻辑通过调用 Colleague1 和 Colleague2 的行动方法 action()将这个信息传给所有有关的同事对象。具体的活动时序如下图所示。



最后指出的是，在上面的设计中，当调停者对象收到某一个同事对象发生了变化通知时，会逐一通知所有的同事对象，这其中包括作为事件起源的那个同事对象。在上面的示意性系统中，这个对象就是 Colleague1。在一个复杂的系统中，调停者在事件发生时不一定会通知所有的同事对象，而仅仅会通知有关的同事对象。

那么哪些同事对象是与之有关的呢？这是一个与系统的商业逻辑有关的问题，读者可以根据自己所面对的系统的具体需要决定哪些对象是有关对象，哪些对象是无关对象；哪些对象应该通知，哪些对象不该通知。

55.3　调停者模式的实现

具体同事对象的创建

在很多情况下，这些具体同事对象均是由一个对象创建的，这个对象也可以把调停者类作为内部类。这样的做法使系统更加强壮。

具体调停者对象的内部状态

在决定实现调停者模式的时候，设计师还要做出一些重要的决定。一个重要的决定就是谁应当负责维护调停者对象的内部状态。

一般而言，可以选择同事对象维护所有的具体调停者对象的内部状态，也可以让每一个具体调停者对象自己维护自己的内部状态。

是否使用抽象调停者

如果非常肯定具体调停者角色只有一个的话，可以省略抽象调停者类。如果同时存在多个具体调停者角色的话，就有可能会在这些具体调停者角色中发现共同的代码。这样一来，就需要一个共同的类型，并将重复的代码移动到超类中去。

同事类如何与调停者通信

同事对象可以将它们自身作为参量传给调停者对象。

55.4 迪米特法则（LoD）

迪米特法则说："只与你直接的朋友们通信"。迪米特法则要求每一个对象与其他对象的相互作用均是短程的，而不是长程的。而且只要可能，朋友的数目越少越好。换言之，一个对象只应当知道它的直接合作者的接口。关于迪米特法则的详细讲解，请见本书"迪米特法则（LoD）"一章。

调停者模式创造出一个调停者对象，将系统中有关的对象所引用的其他对象数目减到最少，使得一个对象与其同事的相互作用被这个对象与调停者对象的相互作用所取代。显然，调停者模式是迪米特法则的一个具体应用。

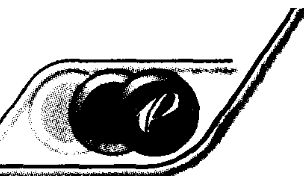
本章开始时所讨论的各个贸易地区加入 WTO 的例子，便是一个从不遵守迪米特法则，通过改造达到满足迪米特法则要求的例子。

55.5 调停者模式的优点和缺点

调停者模式的优点如下：

（1）适当使用调停者模式可以较少使用静态的继承关系，使得具体同事类可以更加容易地被复用。

（2）适当使用调停者模式可以避免同事对象之间的过度耦合，使得调停类与同事类可以相对独立地演化。



（3）调停者模式将多对多的相互作用转化为一对多的相互作用，使得对象之间的关系更加易于维护和理解。

（4）调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

调停者模式的缺点如下：

（1）调停者模式降低了同事对象的复杂性，代价是增加了调停者类的复杂性。当然，在很多情况下，设置一个调停者并不比不设置一个调停者更好。

（2）调停者类经常充满了各个具体同事类的关系协调代码，这种代码常常是不能复用的。因此，具体同事类的复用是以调停者类的不可复用为代价的。

显然，调停者模式为同事对象，而不是调停者对象提供了可扩展性，所以这个模式所提供的可扩展性是一种（向同事对象）倾斜的可扩展性。

55.6　调停者模式的使用条件

大多数对模式的研究集中于模式应当在什么情况下使用，却往往忽视这些模式不应当在什么情况下使用。调停者模式是常常被滥用的几个模式之一，但与其他模式不同的是，调停者模式常常在一个已经很糟的设计中使用。

本节就讨论一下在什么情况下不应当使用调停者模式。

不应当在责任划分混乱时使用

通常的情况下，一个初级设计师在对面向对象的技术不熟悉时，会使一个系统在责任的分割上发生混乱。责任分割的混乱会使得系统中的对象与对象之间产生不适当的复杂关系。

这时候，一个很糟的想法就是继续这个错误，并使用调停者模式“化解”这一团乱麻。实际上，这样一来，责任错误划分的混乱不但不会得到改正，而且还会制造出一个莫名其妙的怪物：一个处于一团乱麻之中的混乱之首。

不应当对“数据类”和“方法类”使用

初级设计师常常会设计出这样的一种系统：让一系列类只含有数据，另一些类只含有方法。比如，描述一个客户时，这些设计师首先设计出一个叫做客户数据的类，只含有客户数据；另外再设计一个类叫做“管理类”，含有操作客户以及此客户购买公司产品、付账的方法。管理类自然会涉及到其他的类，诸如产品数据类、订单数据类、付账数据类、应收账数据类等。

如此一来，“管理类”就变成一个操作所有的数据类的方法类。这是很没有道理的设计，但是有些设计师错误地把这叫做“调停者模式”的应用。

正确理解封装

封装首先是行为，以及行为所涉及的状态的封装。行为与状态是不应当分割开来的。

调停者模式的用途是管理很多的对象的相互作用，以便使这些对象可以专注于自身的行为，而独立于其他的对象。上面的“管理类”则不同，因为它把一个数据对象的内部状态取出来，在运算后，再插入到另一个数据对象里面。这些数据对象根本没有包装任何的行为，是被动的行为接收者。显然，这与调停者模式根本不相干。

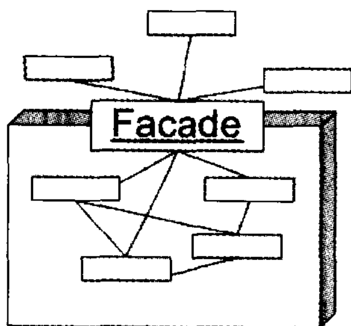
正确的做法是让每一个类对自己的行为负责，纯数据类的划分是完全没有道理的。

在一些特殊情况下，将操作封装起来是好的设计方案，比如策略模式以及访问者模式都是将操作封装起来的设计。但是在这两种模式使用的时候，并不需要调停者对象。其次，这些封装了操作的对象同时也封装了操作所涉及的内部状态，而没有割裂这些状态。

55.7 与调停者模式有关的模式

门面模式

门面模式和调停者模式很相似，两者均用来给出一个低耦合度的系统。门面模式为一个子系统提供一个简单的接口，其中消息的传送是单方向的，因为门面模式的客户端只通过门面类向子系统发出消息，而不是相反的情况。门面模式的结构图如下图所示。



调停者模式则不同，它与同事对象的相互作用是多方向的。

观察者模式

调停者模式与观察者模式是功能相类似的设计模式，它们之间是相互竞争的关系。换言之，使用了其中一者往往意味着放弃另一者。观察者模式通过引入观察者对象和主题对象来达到将通信分散化的目的；而调停者模式则封装了对象之间的通信，从而将通信集中到一个个中介对象中。

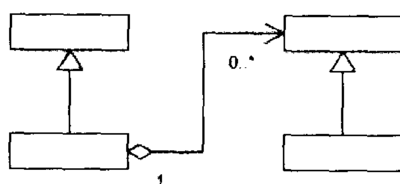
观察者模式需要观察者对象和主题对象的相互协作才能达到目的，而且一个观察主题

对象通常有几个观察者对象，而一个观察者对象也可以同时观察几个主题对象。

由于观察者模式将性能分散到几个对象中，因此更容易达到复用的目的。但是由于每一次通信都涉及几个对象，因此，使用观察者模式的设计可能不容易读懂。使用了调停者模式使得系统的通信都要经过这个中介对象，在所涉及的同事对象数目不多的情况下，使用调停者模式的设计是比较容易懂的设计。

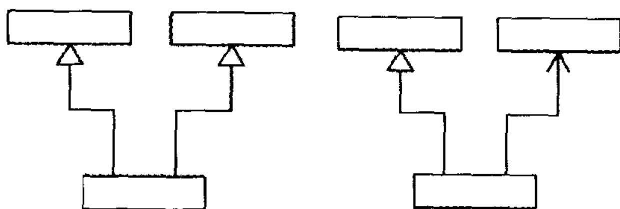
在观察者模式中，观察者与被观察者是不同的角色；而在调停者模式中，所有的同事对象都扮演同样的角色。

大家知道，观察者模式已经被 Java 语言选做 AWT 的事件处理模式。而由于调停者模式并不区分施予者与接收者，且过度将事件处理集中化，所以调停者模式不适合用于设计一个大型系统的事件处理机制。观察者模式的简略类图如下图所示。



适配器模式

调停者对象经常使用适配器对象接收同事对象的状态变化。适配器模式的简略类图如下图所示，左边的是类的适配器模式，右边的是对象的适配器模式



附录 A　设计模式一览表

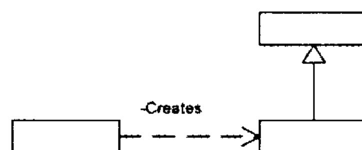
A.1　创 建 模 式

创建模式（Creational Pattern）是对类的实例化过程的抽象化。

简单工厂（Simple Factory）模式

简单工厂模式是类的创建模式，又叫做静态工厂方法（Static Factory Method）模式。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。

简单工厂模式的简略类图如右图所示。

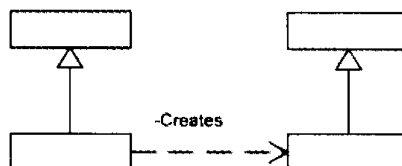


工厂方法（Factory Method）模式

工厂方法模式是类的创建模式，又叫做虚拟构造子（Virtual Constructor）模式或者多态性工厂（Polymorphic Factory）模式。

工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。

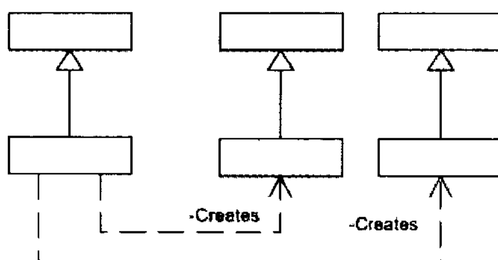
工厂方法模式的简略类图如右图所示。



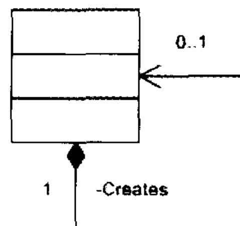
抽象工厂（Abstract Factory）模式

作为对象的创建模式，抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象。这就是抽象工厂模式的用意。

抽象工厂模式的简略类图如下图所示。



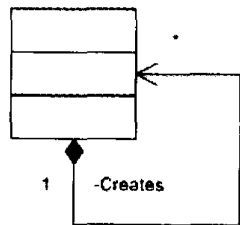
单例（Singleton）模式



作为对象的创建模式，单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

单例模式的简略类图如右图所示。

多例（Multiton）模式



作为对象的创建模式，多例模式或多例类有以下的特点：

（1）多例类可以有多个实例；

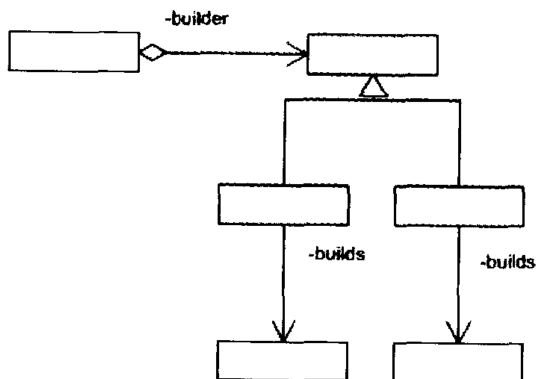
（2）多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

多例模式的简略类图如右图所示。

建造（Builder）模式

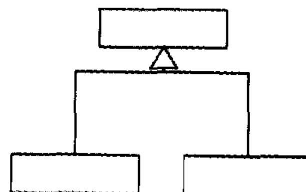
建造模式是对象的创建模式。建造模式可以将一个产品的内部表象与产品的生成过程分割开来，从而使一个建造过程可以生成具有不同的内部表象的产品对象。

建造模式的简略类图如下图所示。



原始模型（Prototype）模式

原始模型模式属于对象的创建模式。通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象，这就是原始模型模式的用意。原始模型模式的简略类图如右图所示。



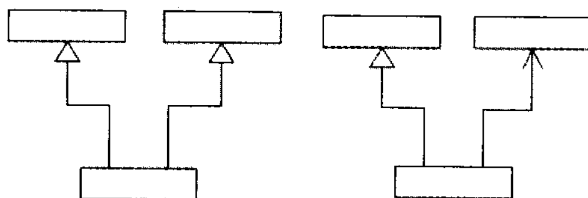
A.2 结构模式

结构模式（Structural Pattern）描述如何将类或者对象结合在一起形成更大的结构。

适配器（Adapter）模式

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

如下图所示，左边是类的适配器模式的简略类图，右边是对象的适配器模式的简略类图。

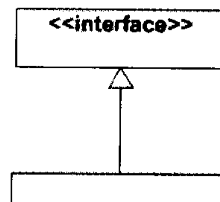


缺省适配（Default Adapter）模式

缺省适配模式为一个接口提供缺省实现，这样子类型可以从这个缺省实现进行扩展，而不必从原有接口进行扩展。

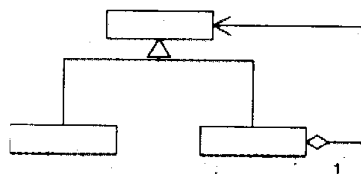
作为适配器模式的一个特例，缺省适配模式在 Java 语言中有着特殊的应用。

缺省适配模式的简略类图如右图所示。



合成（Composite）模式

合成模式属于对象的结构模式，有时又叫做部分－整体（Part-Whole）模式。合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式可以使客户端将单纯元素与复合元素同等看待。合成模式的简略类图如右图所示。

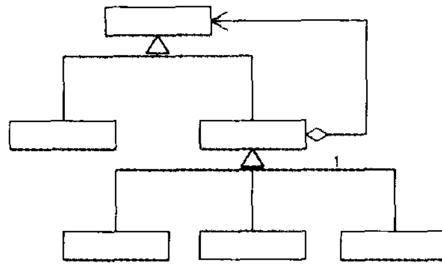


装饰（Decorator）模式

装饰模式又名包装（Wrapper）模式，是对象的结构模式。

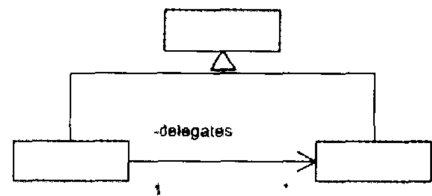
装饰模式以对客户端透明的方式将一个对象的功能加以扩展。装饰模式为功能的扩展提供了一个除继承关系之外的另一个可选方案。装饰模式的简略类图如下图所示。





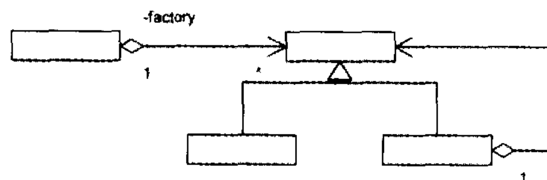
代理（Proxy）模式

代理模式是对象的结构模式。代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。代理模式的简略类图如右图所示。



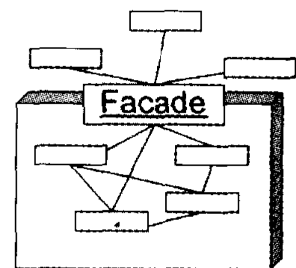
享元（Flyweight）模式

享元模式是对象的结构模式。享元模式以共享的方式高效地支持大量的细粒度对象。享元模式的简略类图如下图所示。



门面（Facade）模式

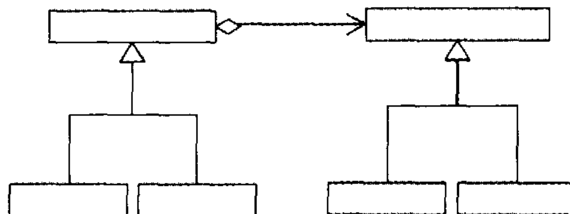
门面模式是对象的结构模式。外部与一个子系统的通信必须通过一个统一的门面（Facade）对象进行，这就是门面模式。门面模式提供一个高层次的接口，使得子系统更易于使用。门面模式的简略类图如右图所示。



桥梁（Bridge）模式

桥梁模式是对象的结构模式，又称为柄体（Handle and Body）模式，或接口（Interface）模式。

桥梁模式将抽象化（Abstraction）与实现化（Implementation）脱耦，使得二者可以独立地变化。桥梁模式的简略类图如下图所示。



A.3 行为模式

行为模式（Behavioral Pattern）是对不同的对象之间划分责任和算法的抽象化。

不变（Immutable）模式

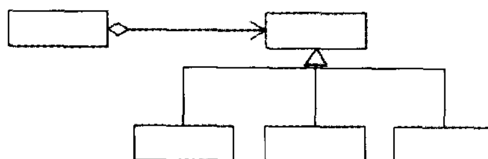
一个对象的状态在对象被创建之后就不再变化，这就是所谓的不变模式。（图略）

策略（Strategy）模式

策略模式属于对象的行为模式，又称做政策（Policy）模式。

策略模式的用意是：准备一组算法，并将每一个算法封装到一个独立的具体类中。这些具体类具有共同的接口，从而使得它们可以互换。策略模式使得算法可以在不影响到客户端的情况下发生变化。

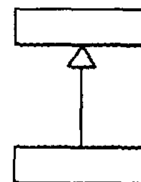
策略模式的简略类图如下图所示。



模版方法（Template Method）模式

模版方法模式是类的行为模式。

准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模版方法模式的用意。模版方法模式的简略类图如右图所示。

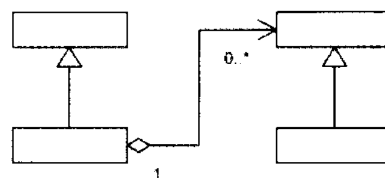


观察者（Observer）模式

观察者模式是对象的行为模式，又叫做发布-订阅（Publish/Subscribe）模式、模型-

视图（Model/View）模式、源–监听器（Source/Listener）模式或从属者（Dependents）模式。

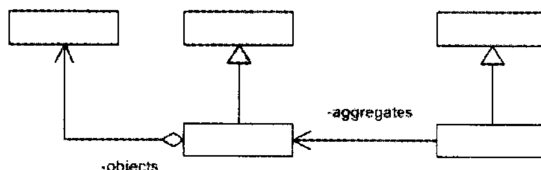
观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。观察者模式的简略类图如右图所示。



迭代子（Iterator）模式

迭代子模式又叫游标（Cursor）模式，是对象的行为模式。迭代子模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部表象。

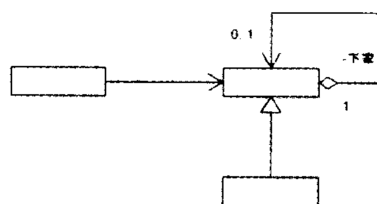
迭代子模式的简略类图如下图所示。



责任链（Chain of Responsibility）模式

责任链模式是一种对象的行为模式。

在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。责任链模式的简略类图如下图所示。

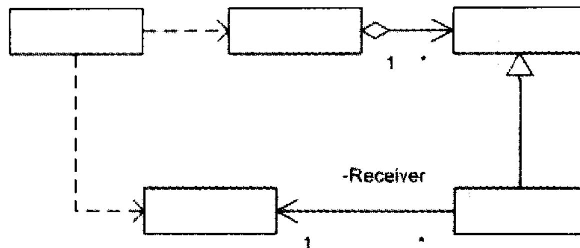


命令（Command）模式

命令模式属于对象的行为模式。命令模式又称为行动（Action）模式或交易（Transaction）模式。

命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志。可以提供命令的撤销和恢复功能。

命令模式的简略类图如下图所示。

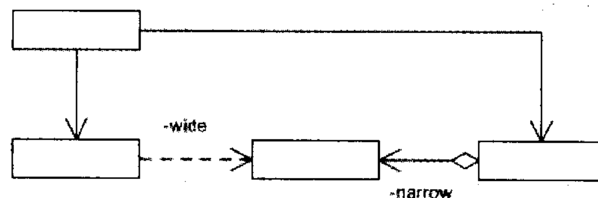


备忘录（Memento）模式

备忘录模式又叫做快照模式（Snapshot Pattern）或 Token 模式，是对象的行为模式。

备忘录模式的用意是：在不破坏封装的条件下，将一个对象的状态捕捉住，并外部化存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

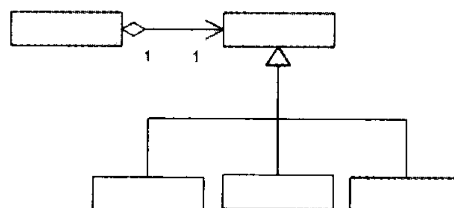
备忘录模式的简略类图如下图所示。



状态（State）模式

状态模式又称为状态对象模式（Pattern of Objects for States），是对象的行为模式。

状态模式允许一个对象在其内部状态改变的时候改变其行为。这个对象看上去就像是改变了它的类一样。状态模式的简略类图如下图所示。

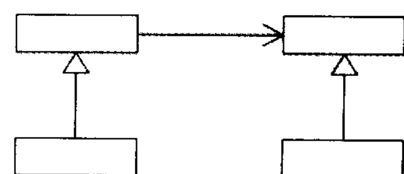


访问者（Visitor）模式

访问者模式是对象的行为模式。

访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接收这个操作的数据结构则可以保持不变。

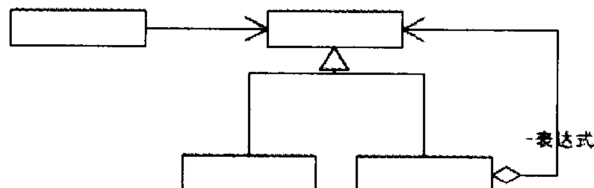
访问者模式的简略类图如右图所示。



解释器（Interpreter）模式

解释器模式是类的行为模式。

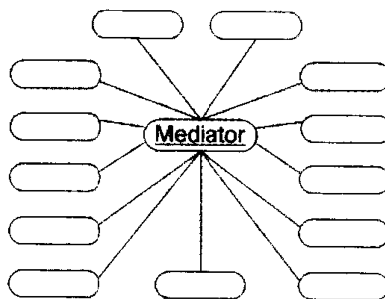
给定一个语言之后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。解释器模式的简略类图如下图所示。



调停者（Mediator）模式

调停者模式是对象的行为模式。

调停者模式包装了一系列对象相互作用的方式，使得这些对象不必互相明显引用，从而使它们可以较松散地耦合。当这些对象中的某些对象之间的相互作用发生改变时，不会立即影响到其他的一些对象之间的相互作用，从而保证这些相互作用可以彼此独立地变化。调停者模式的简略类图如下图所示。



附录 B　UML 图标及其 Java 实现一览表

为了方便读者阅读本书，这里特地给出本书中最常用到的 UML 图标及其在 Java 语言中的实现。

UML 图标	Java 语言中的实现
对象（Object）	
Object : MyClass	（无）
类（Class）	
Programmer -employeeId +increaseSalary()	public class Programmer { private int employeeId; public void increaseSalary() { ... } }
接口（Interface）	
<<interface>> MyInterface +MyMethod()	public interface MyInterface { public void MyMethod(); }
库（Package）	
com.xyzinc.beans	package com.xyzinc.beans;
依赖关系（Dependency）	
— — —>	关系对象出现在局域变量或者方法的参量里；或者关系类的静态方法被调用等
Programmer +design(in x : System) - - - -> System	public class Programmer { public void design(System x) { .... } }
关联关系（Association）	
———>	关系对象出现在实例变量中
Programmer -ts : Timesheet +keep() ——> Timesheet	

（续表）

聚合关系（Aggregation）	
◇——→	关系对象出现在实例变量中
Programmer -type : EmployeeType ◇——→ 1 0..* EmployeeType	public class Programmer { private EmployeeType type; .... }
合成关系（Composition）	
◆——→	关系对象出现在实例变量中
Programmer -ts : Timesheet +keep() ◆——→ 1 0..* Timesheet	public class Programmer { private Timesheet ts; public void keep() { .... } }
推广关系（Generalization）	
◁——	使用 extends 关键字
Programmer ◁—— JavaProgrammer	public class Programmer { .... } public class JavaProgrammer extends Programmer { .... }
实现关系（Realization）	
◁- - - -	使用 implements 关键字
<<interface>> Professional ◁- - - - Programmer	public Interface Professional { .... } public class Programmer implements Professional { .... }

注意

（1）关联关系、聚合关系和合成关系不能够从 Java 语法上表现出来。

（2）关联关系常常可以进一步说明为聚合关系或者合成关系，因此上面关联关系与合成关系的实现是一样的。

附录 C　中英术语对照表

A
- abstract class（抽象类）
- abstract factory（抽象工厂）
- Abstract Factory Pattern（抽象工厂模式）
- abstract operation（抽象操作）
- Abstract Syntax Tree（抽象语法树，简称 AST）
- abstraction（抽象）
- access modifier（存取权限）
- accessor methods（取值方法）
- adapter（变压器，适配器）
- Adapter Pattern（适配器模式）
- aggregation（聚合关系）
- Applet（小程序，不翻译）
- application（应用程序）
- Application Programming Interface（应用程序编程接口，API）
- Application Server（应用服务器）
- architecture（构架）
- association（关联关系）
- asynchronous（非同步的）
- attribute（属性）

B
- Bean（JavaBean）
- behavior（行为）
- Behavioral Patterns（行为模式）
- binding（绑定）
- bridge（桥梁）
- Bridge Pattern（桥梁模式）
- builder（建造者）
- Builder Pattern（建造模式）

C
- call（调用）
- chain of responsibility（责任链）
- Chain of Responsibility Pattern（责任链模式）
- checkpoints（检查点）
- child（子）

- class（类）
- class diagram（类图）
- class hierarchy（类分层结构）
- class library（类库）
- class method（类方法）
- class pattern（类的模式）
- client/server（客户机/服务器）
- client（客户端）
- collaboration diagram（协作图）
- collaboration（协作）
- collection（聚集）
- command（命令）
- Command Pattern（命令模式）
- Component Object Model（构件对象模型或者 COM）
- Common Object Request Broker Architecture（简称为 CORBA）
- compile time（编译时期）
- component diagram（构件图）
- component model（构件模型）
- component（构件）
- composite（合成）
- composite aggregation（合成聚合关系）
- composition（合成）
- Composite Pattern（合成模式）
- concrete class（具体类）
- concrete（具体）
- concurrency（并发）
- configuration（配置）
- constructor（构造函数）
- container（容器）
- context（环境）
- Cookie（小甜饼文件，不翻译）
- Creational Pattern（创建模式）

D

- decorator（装饰）
- Decorator Pattern（装饰模式）
- Database Management System（数据库管理系统，DBMS）
- database（数据库）
- datatype（数据类型）

- delegation（委托）
- Delegation Event Model（委派事件模型，简称 DEM）
- deliverable（可交付的）
- dependency（依赖关系）
- depickling（回鲜）
- deployment diagram（部署图）
- deployment（部署）
- deserialize（并行化）
- deserialization（并行化）
- design pattern（设计模式）
- design（设计）
- developer（开发人员）
- diagram（图）
- dispatch（分派）
- document（文档）
- double-byte character set（双字节字符集，DBCS）
- double dispatch（双重分派）

E

- e-Business（电子商务）
- encapsulation（封装）
- Enterprise JavaBean（EJB）
- entitlement（资格，权限）
- entity class（实体类）
- enumeration（枚举）
- environment（环境）
- event（事件）
- expression（表达式）
- extend-relationship（扩展关系）
- extend（扩展）
- extensible（可扩展）
- extensibility（可扩展性）
- external state（外蕴状态）
- exception（异常）

F

- facade（门面）
- Facade Pattern（门面模式）
- Facade Proxy Pattern（门面代理模式）
- factory（工厂）
- Factory Method Pattern（工厂方法模式）
- Factory Pattern（工厂模式）
- feature（特性）

- field（字段）
- final state（最终状态）
- flatten（串行化）
- flyweight Pattern（享元模式）
- focus of control（控制焦点）
- fragile（易碎的）
- fragility（易碎性）
- framework（框架）

G

- generalization（泛化关系）
- granularity（粒度）
- graphical user interface（图形用户界面，GUI）
- guard condition（警戒条件）

H

- hierarchy（等级结构）
- Hypertext Markup Language（超文本标记语言，HTML）

I

- implementation（实施）
- immutable（不变的）
- Immutable Pattern（不变模式）
- index（指标）
- instance（实例）
- instance variable（实例变量）
- internal state（内蕴状态）
- interpreter（解释器）
- Interpreter Pattern（解释器模式）
- iteration（迭代）
- iterator（迭代子）
- Iterator Pattern（迭代子模式）

J

- Java（Java）
- Java Development Kit（Java 开发工具包，JDK）
- Java Foundation Classes（Java 基础类，JFC）

K

- keyword（关键字）

L

- listener（监听器，监听程序）
- logical view（逻辑视图）

M

- Mail Exchange（邮件交换，或简称 MX）
- maintain（维护）
- maintainability（可维护性）
- mechanism（机制）
- mediator（调停者）
- Mediator Pattern（调停者模式）

- memento（备忘录）
- Memento Pattern（备忘录模式）
- message（消息）
- messaging（消息传递）
- method call（方法调用）
- method（方法）
- Model View Controller（模型–视图–控制器，MVC）
- module（模块）
- multi-dispatch（多分派）
- multiple dispatch（多重分派）
- multiple inheritance（多重继承）
- multiton（多例）
- Multiton Pattern（多例模式）
- mutable（可变的）
- mutator（改值方法）
- MX Records（邮件交换记录）

N

- n-ary association（多元关联关系）
- naming-directory service（命名–目录服务）
- node（节点）

O

- object（对象）
- object diagram（对象图）
- object-oriented programming（面向对象程序设计，OOP）
- object pattern（对象的模式）
- observer（观察者）
- Observer Pattern（观察者模式）
- operation（操作）
- output（输出）

P

- package（包）
- parameter（参数）
- parent class（父类）
- parent（父）
- pattern（模式）
- pickling（腌渍）
- plug（插入）
- pluggable（可插入）
- pluggability（可插入性）
- polymorphism（多态性）
- private（私有）
- primitive method（基本方法）

- project（项目）
- property（特征）
- protected（保护）
- protocol（协议）
- prototype（原型）
- Prototype Pattern（原型模式）
- proxy（代理）
- Proxy Pattern（代理模式）
- Proxy Facade Pattern（代理门面模式）
- pseudo-state（伪状态）

Q

- quality（质）
- QWAN（Quality Without a Name，无名的质）

R

- rationale（理由）
- RDBMS（关系数据库管理系统）
- read only （只读）
- receiver（接收方）
- reception（接收）
- Refactor（重构）
- reference（引用）
- refinement（改进）
- relationship（关系）
- Remote Method Invocation（远程方法调用，RMI）
- reusable（可复用的）
- reusability（可复用性）
- reuse（复用）
- rigid（刚性的，僵硬的）
- rigidity（刚性，僵硬度）
- RMI（远程方法调用）
- robust（强壮的）
- robustness（强壮性）
- role（角色）
- run time（运行时期）

S

- sender（发送端）
- sequence diagram（序列图）
- serialize（串行化）
- serialization（串行化）
- server（服务器）
- Servlet（服务器小程序，不翻译）
- singleton（单例）



- Singleton Pattern（单例模式）
- software architecture（软件构架）
- snapshot（快照）
- Snapshot Pattern（快照模式）
- spell（念咒语）
- state machine（状态机）
- statechart diagram（状态图）
- state（状态）
- State Pattern（状态模式）
- strategy（策略）
- Strategy Pattern（策略模式）
- string（字符串）
- structural feature（结构特性）
- structural model aspect（模型的结构侧重面）
- Structural Pattern（结构模式）
- subclass（子类）
- superclass（超类）
- supplier（提供端）
- synchronized（同步化的）
- synchronous（同步的）
- system（系统）

T

- template（模版）
- Template Method Pattern（模版方法模式）
- thread（线程）
- timer（定时器）
- transaction processing（事务处理）
- transaction（事务）
- transient object（临时对象）
- transition（产品化/转移）
- traverse（遍历）
- type（类型）

U

- Unified Modeling Language（统一建模语言，UML）
- uni-dispatch（单分派）
- use-case diagram（用例图）

V

- value（值）
- variable（变量）
- view（视图）
- virtual machine（虚拟机，VM）
- viscosity（黏度）

- visibility（可见性）
- visitor（访问者）
- Visitor Pattern（访问者模式）

W

- web application（Web 应用程序）
- web browser（Web 浏览器）
- web server（Web 服务器）
- web site（Web 站点）
- web system（Web 系统）
- Widget（窗口组件）

X

- XML（可扩展标识语言，本书仍然简称为 XML）